Type script Version 2

Fait le 21-11-2017

Table des matières

Les variables : 3

Déclaration Explicite des variables 3

Variable de type tableau 3

Variable de type tuple : 4

Variable de type enum 4

Variable de type any: 4

Fonction 4

Objet 5

alias 5

Union types 5

Check type with typeof 5

Never 5

Nullable type 6

Exercice BankAccount 6

SourceMap property de fichier tsconfig.ts 7

noImplicitAny property 7

Compiler is more smart 7

Nouvelles fonctions propose par Javascript6 8

let & const 8

Block scope 8

Default parameters 8

Rest & Spread 9

Classe et Héritage 9

Destructuring 9

Template literals 9

Exercice ES6 10

Getter et Setter 12

Propriété et méthode statique 12

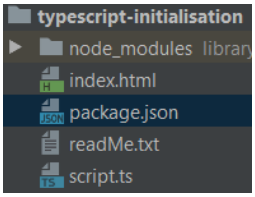
Classe abstract 13

Namespace 14

Namespaces dans différents fichiers 14

Les imports de namespace 16
Plus dans un namesapce 16
Les limites de namespace 16
Modules 16
Interface 17
Interface pour une méthode 18
Interface et propriétés de l'objet 18
Interface et méthodes 19
Interface et classes 19
Interface et classes 19
Interface et l'héritage 20
Qu'est ce qui ce passe pour les interfaces après la compilation 20
Generic 20
Création d'une fonction générique 21
Les tableaux et le type générique 21
Utilisation des types génériques 21
Création des classes génériques 22
Decorator 23
Création de Decorator des classes 23
Decorator Factory 23
Useful Decorator 24
Utiliser plusieurs decorators 24
Decorator pour une méthode 25
Decorator pour les paramètres des méthodes 26
Decorator pour les property 26

Pour installer typescrit : executer la commande : **npm -g install typescrit** c'est le compilateur qui va compiler .ts et générer le ficher .js lisible par le navigateur.
Pour initialiser un projet de type Typescript il fait exécuter dans l'invite de commande :
tsc –init



Dans le dossier de projet exécuté la commande **npm install lite-server --save**
Elle permet d'avoir un serveur http et recharger le contexte à chaque nouveau dev
Dans le fichier package.json il faut ajouter dans l'objet script

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "lite-server"
},
```

On démarre le service de lite-server , pour executer le server : **npm start**
Le compilateur de TS génère un fichier de configuration : **tsconfig.json**
Pour compiler tous les fichiers .ts à la fois , on initialise la configuration de compilateur ts par : **tsc - -init ,** qui va créer un fichier **tsconfig.json .P**our compiler on exécute seulement commande **tsc sans le nom** de fichier
Et bien sur lancer le lite-server par la commande npm start pour avoir les modifications à chaud et la compilation de fichier .ts par la commande **tsc dans l'invite de commande**
Pour compiler les fichiers .ts en exécute la commande **tsc nomDuFichier.ts** ou en exécute seulement tsc et le compilateur compile tous les fichiers .ts

TypeScript est un langage typé si on met le code suivant le compilateur va déclarer une erreur d'affectation, en JavaScripts les variables sont interprétées

```
// string
let myName = 'Maximus';
// myName = 39 ;

// number
let myAge = 34 ;
// myAge = 'Maximus';

// boolean
let hasAdmin = false ;
// hasAdmin = 1
```

Les variables :

Déclaration Explicite des variables

Il est possible de déclarer implicitement les types de variable ou explicitement
La déclaration implicite déclare les variables avec un type **any** la variable pour avoir des valeurs de type objet ou primitives

```
// string déclaration  explicite
let myName:string = 'Maximus';
// myName = 39 ;

// number déclaration  explicite
let myAge:number = 34 ;
// myAge = 'Maximus';

// boolean déclaration  explicite
let hasAdmin:boolean = false ;
// hasAdmin = 1

// déclaration  implecite
let myRealAge ;
myRealAge = 34;
myRealAge = '34';
```

Variable de type tableau

```
let voitures = ['MERCEDES', 'BMW', 'MAZERATI'];
// initialiser le tableau par des numéro
voitures =[1 , 2 , 3] ; // problème de type de tableau  Type 'number' is not
assignable to type 'string'.
console.log(typeof voitures); // on recuper un resultat = object

// pour résoudre le problème dans ce cas on utilise la déclaration implicite
let voitures: any[] = ['MERCEDES', 'BMW', 'MAZERATI'];
voitures =[1 , 2 , 3] ; // le tableau accepte diffèrent type de valeur
```

Variable de type tuple :

```
let moteur = ['mercedes v8 bi-compressor', 550];
// moteur est de type any par defaut
// tuples
// si la variable a le même format quand en le set
// on peut crée une variable de type tuples avec une structure bien defini
 let moteur:[string, number] = ['mercedes v8 bi-compressor', 550];
```

```
voitureTuple ==> ▼(2) [1, "BMW"]
                   0: 1
                   1: "BMW"
                   length: 2
                 ▶ __proto__: Array(0)
```

Variable de type enum

```
/**
 * enum creer un enumerateur
 */
enum Fruit{
    Banane, //0 le compilateur assigne des nombres incrementé commence par 0
    Orange = 200, //1   on peut changer la valeur retour
    Pomme ,  //2  le compilateur n'accepte pas des valeur de type string
    Kiwi= 3 ,    //3
    Poire
}
let myFruit : Fruit = Fruit.Kiwi;
console.log('Banane',Fruit.Banane , 'Orange',Fruit.Orange ,'pomme' , Fruit.Pomme
,'kiwi',Fruit.Kiwi ,'poire',Fruit.Poire);
```

```
21:54:12.047 Banane 0 Orange 200 pomme 201 kiwi 3 poire 4
```

Variable de type any:

```
let voiture  = "Alfa romeo" ;
console.log('Voiture: ' , voiture) ;
// typescript a créé la voiture de type string
// la réaffectation de la variable par un autre type n'est pas accepté
voiture ={
    marque:"MERCEDES",
    moteur:"DIESEL"
};
console.log('Voiture: ' , voiture) ;
// pour résoudre ce problème en change le type de la variable voiture par any

let voiture:any  = "Alfa romeo" ;
```

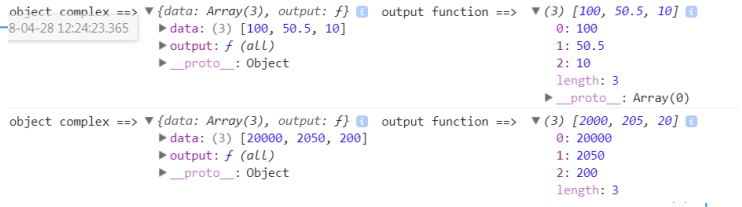
```
top   Filter
1
Voiture:  Alfa romeo
Voiture:  ▼{marque: "MERCEDES", moteur: "DIESEL"}
            marque: "MERCEDES"
            moteur: "DIESEL"
          ▶ __proto__: Object
>
```

Fonction

```
// functions
function returnName() :string{   return'Ralf' }
//void
function sayHello():void{    console.log("Hello! ") }
// arguments types
function multiply(value1 , value2) :number{     return value1*value2; }
/* les arguments de la fonction multiply ne sont pas type donc il sont de type any , si on execute la fonction on
rerupere une valeur NaN => not a number  c'est logilt 1 *'Ralf' erreur*/
console.log(multiply(1,'Ralf'));
/* pour resoudre ce probleme on va typé les paramere
* le compilateur detecte l'erreur dés la compilation et non au moment de l'execution*/
function multiplyType(value1:number , value2:number) :number{     return value1*value2; }
function multiplyType10(value1:number , value2:number) :number{   return value1*value2*10; }
console.log(multiply(1,1));
// function as type
let fnMultiply  ; // type any
fnMultiply = sayHello;
fnMultiply();
fnMultiply = multiplyType ; // ref vers la function
console.log('----'+fnMultiply(5,2));
/* fnMultiply est de type any elle accepte tous les type , pour que fnMultiply soit un type function ,on lui affect
au debut un interface fonctionnel avec les memes parametre de la function le comportement est defini dans la
function   */
let fnMultiply2 :(a:number , b: number) => number ; // ici on typé la varible
fnMultiply2 = multiplyType ; // ici on specifi l'implementation de comportement par la function
console.log('comportement avec  multiplyType',fnMultiply2(2,5));
fnMultiply2 = multiplyType10 ;// c'est transparent car on passe par une interface fonctionnel pour la definition de
la varible
console.log('comportement avec  multiplyType10',fnMultiply2(2,5));
```

```
----10
comportement avec  multiplyType 10
comportement avec  multiplyType10 100
```

Objet




```
//object
let userData : {name:string , age:number} ={
    name:'Youssef' ,
    age:40
}
/*on peut definir l'objet avec sa partie declarative
ou type definition {name:string , age:number}
et par sa valeur apres le signe egale  = {cc:'' ,bb:'' }
 */
// userData ={
//     a:'toto'
//     b:45
// };
/* si on assige a l'objet userData des propriété qui differe de sa definition
* le comilateur nous dit que c'est pas compatible */

// complex object
let complex: { data :number[] , output:(all:boolean) => number[]} =
    {
        data : [100 , 50.5 , 10] ,
        output :function(all:boolean):number[]{
            return this.data ;
        }
    };
console.log('object complex ==>' , complex , 'output function ==> ', complex.output(true));
complex= {
    data : [200 , 20.5 , 2] ,
    output :function(all:boolean):number[]{
        this.data.forEach((e , index)=> { this.data[index] = e*10 ;});
        return this.data  ;
    }
}
console.log('object complex ==>' , complex , 'output function ==> ', complex.output(true));
```

alias

```
// type alias
/* enregistre un type dans un mots clé , l'utilisé comme type pour définir d'autre object
 * pour changer la définition on change seulement sur l alias
 * pour appliquer la modification sur toutes les variables de ce type
 * il ya aussi les classe pour definir la structure de l'object */
type Complex = { data :number[] , output:(all:boolean) => number[]} ;
let complex2 : Complex ;
complex2 = {
    data : [1 , 1.5 , 2] ,
    output :function(all:boolean):number[]{
        return this.data ;
    }
};
```

Union types

```
// union types
// definier une variable avec un ensemble de type
let myRealAgeNow :number | string = 40 ;  // on accepte le type number ou string
myRealAgeNow = '40';
//myRealAgeNow = true ; les valeurs de type boolean ne seront pas acceptés ,
```

Check type with typeof

```
// check type
let testTypeValue = 30;
if(typeof testTypeValue == 'number' ){
    console.log("testTypeValue is a number ");
}
```

Never

```
// never
/* type never permet de dire au compliateur que une fonction ne renvoie jamais une valeur
* never est plus puissante que void , au moment ou la fonction peut envoyer une exception
* avec void le compilateur catch cette exception mais avec never la fonction ne renvoie jamais
une valeur*/
function neverReturns():never{
    throw new Error('An error!') ;
}
```

Nullable type

```
// nullable type
/* pour que le compilateur accepte la valeur null ,il faut configuer dans le fichier tsconfig.ts l'objet
compilerOptions.strictNullChecks = false  si non il faut specifier pendant la déclaration de la variable une
union des type ex : canBeNull : number | null
*/
let canBeNull : number | null = 7 ; // on accepte le type numbre et null
canBeNull = null ;
let canAlsoBeNull; // la variable  n'est pas typé donc elle a une valeur undefined
console.log("canAlsoBeNull", canAlsoBeNull);
canAlsoBeNull = null ;
let canThisBeAny : number | null = null ;
canThisBeAny = 12;
```

Le fichier rsconfig.ts

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "noImplicitAny": false,
    "sourceMap": false,
    "strictNullChecks":true
  },
  "exclude": [
    "node_modules"
  ]
}
```

Exercice BankAccount

```
// exercice type bank  ecrit en Typescprit
type Bankaccount = { money :number , desposit:(value:number)=>void} ;
type Myself = { name:string , bankAcount:Bankaccount , hobbies : string[]};

let bankAccount : Bankaccount = {
    money:2000 ,
    desposit:function(value:number):void{
        return this.money +=value;
    }
};

let myself :Myself = {
    name:"Ralf",
    bankAcount:bankAccount,
    hobbies:["Sports","Cooking"]
};

myself.bankAcount.desposit(3000);
console.log(myself);
```

```
// Format javascript
let bankAccount = {
    money: 2000,
    deposit(value) {
        this.money += value;
    }
};

let myself = {
    name: "Max",
    bankAccount: bankAccount,
    hobbies: ["Sports", "Cooking"]
};

myself.bankAccount.deposit(3000);

console.log(myself);
```

noEmitOnError ne pas créer le fichier .js s'il y a des erreurs dans le fichier .ts

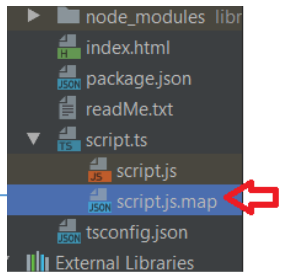
```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "noImplicitAny": false,
    "sourceMap": false,
    "strictNullChecks":true,
    "noEmitOnError": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

Si on ne veut pas generer le fichier .js quand on compile le fichier .ts dans le fichier tsconfig.ts on active l'option **noEmitOnError= true, si** on a des erreus dans les fichiers .ts le compilateur ne genere pas les fichiers .js , par défaut cette fonction est desactivé

SourceMap property de fichier tsconfig.ts

La propieté sourceMap de compilateur permet de créer un fichier de mappin entre le fichier .ts et le fichier .js on mode debbug on peut debbugé avec les poinds d'arret dirrectement sur les fichiers .ts sans passer par les fichier .js generer

```
{
  "compilerOptions": {
    "sourceMap": true,
    "strictNullChecks":true,
```



noImplicitAny property

Par défaut le type des variables est un type any si on le spécifie pas le compilateur implicitement sait que la variable est de type any , pour spécifier explicitement le type any pour une variable on doit configurer le paramètre oImplicitAny = true dans le fichier tsconfig.ts

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "noImplicitAny": true ,
```

Compiler is more smart

```
// compiler is more smart
function controlMe(isTrue :boolean , somthingElse:boolean){
    let result :number ;
    if(isTrue){
        result = 12
    }
    /*on a activer l'option  strictNullChecks :true
     * le compilateur détecte que cette variable renvoie un null si le test if(isTrue) et faut
      * la variable sera pas assigné dans la fonction renvoie un null*/
    result = 33 ;  // pour evité le probleme
    /* on a ajouter aussi dans le fichier tsconfig.ts le parametre noUnusedParameters: true
    * le compilateur detecte que la variable somthingElse est nul part utilisé
    * donc il affichie un probleme de compilation */
    let som:boolean = somthingElse  ;
    /*ces paramètres de compilateur permet d'avoir un code propre*/
    return result ;
}
```

Tsconfig.ts

```
{
  "compilerOptions": {
    "strictNullChecks":true,
    "noEmitOnError": true ,
    "noUnusedParameters": true
```

Pour plus d'information sur le compilateur :

http://www.typescriptlang.org/docs/handbook/tsconfig-json.html
http://www.typescriptlang.org/docs/handbook/compiler-options.html

Nouvelles fonctions propose par Javascript6

let & const

```
// quand on compile  on utilise  tsc -w  ==> -w  en mode watchdog
// let & const
console.log("LET & CONST")
let variable ="Test" ;
console.log(variable);
variable = "Another value";
console.log(variable);

const maxLevels = 100;
console.log(maxLevels);
// maxLevels=99;  le compilateur detecte maxLevels est une constant donc elle est on mode readonly
```

Block scope

```
// Block scope
function reset(){
    //console.log(variable);  la variable=> variable je ne peux pas acceder à celle déclarer hors la methode
reset()
    // pour le faire il faut passer en parametre la valeur de la variable
    // on ne change pas la valeur de la variable meme si elle est declarer avec le meme nom
    //c'est le comportement de Block scope
    let variable = null ;
    console.log("variable declaré dans reset() ==>",variable);
}
reset();
console.log("variable hors methode ==>",variable);

// Arrow functions  fonctions flèche
console.log("ARROW FUNCTION");
// ecrire une methode addNumbers avec l'encienne methode
const addNumbers = function(number1:number , number2:number) :number{
    return number1+number2;
}
console.log("addNumbers(1,2) ==> ",addNumbers(1,2));
console.log("addNumbers(7,3) ==> ",addNumbers(7,3));
// ecrire une methode multiplyNumbers avec la nouvelle methode Arraw function
// si on a dans le comportement de la methode une seule instruction on peut l'ecrire sur une ligne
const multiplyNumbers = (number1:number , number2:number)=> number1*number2;
// ici la comportement de la methode exectute plusieurs instructions donc on met le comportement dans une block
mustash
const addNumbers2 = (number1:number , number2:number)=>{
  const a:number = 10 ;
  return number1 + number2 +a ;
};
console.log("multiplyNumbers(2,3)==> ", multiplyNumbers(2,3));
console.log("addNumbers2(2,3)==> ", addNumbers2(2,3));
/* function Arrow  with no arguements ici on a une fonction fléché sans arguments*/
const helloRalf = ()=> {
  console.log("helloRalf() ==> Hello Ralf!")
};
helloRalf();
// si on a un seul argument c'est possible de ne pas mettre les ()
// et recuperer directement la valeur de la variable la methode , mais il est toujous bien de type la variable
const greetFriend = friend  => console.log("greetFriend() ==> ",friend) ;
greetFriend("TITI");
```

Default parameters

```
/ Default Parameters
console.log("DEFAUTL PRAMANETERS") ;
/*
* on a ajouté une valeur par défaut = 10 au paramètre de la méthode * au moment où on appelle à la méthode sans
paramètre la valeur 10 sera prise , la valeur de end = start-2 , cette syntaxe sera accepter par le compilateur
* si seulement le param end est declaré apres le param start si non c'est une erreur de compilation */
const countdown  =(start:number = 10 ,end :number = start-2):void => {
    console.log("start au début ==> ",start) ;
    while(start> 0){
        start--;
    }
    console.log("start apres ==>" ,start ,"end ==>",end);
}
countdown();
```

Rest & Spread

```
// Rest & Spread
 console.log("REST & SPREAD") ;
 const numbers = [1,100,60,78] ;
 console.log(Math.max(1,100,60,78)) ;
 // ... c'est la fonction Spread de Es6  elle permet de disperser ou éclater les éléments d'un tableau
// sans passer par une boocle
console.log(Math.max(...numbers));
console.log("...numbers ==> " ,...numbers);

/*la fonction Rest permet de récupérer un ensemble de paramètre envoyer à la methode
et de creer un tableau a pratir des valeurs envoyer*/
const makeArray = function(name:string , ...args:number[]){
    console.log(name);
    return args ;
}
// le premier paramètre est le nom , puis avec la fonction rest ==> ...{Var} on crée un tableau
console.log("makeArray ==>" , makeArray("Ralf",1,25,6,2));
```

Destructuring

```
// destructuring   la fonction d'eclatement ou destruction
console.log("DESTRUCTRING");
const myCars =["BMW","ALFA","AUDI","MERCEDES"];
// si on veut récuperer le nom de la voiture dans une variable
let voiture1 = myCars[0];
let voiture2 = myCars[1];
console.log(voiture1,voiture2);
// avec ES6 on a la possibilité d'eclater les valeurs de tableau sur des variables
const [bmw , alfa , audi] = myCars ;
console.log("voiture1 ==>" , bmw , "voiture2 ==>" , alfa , "voiture 3 ==> ", audi);

// destructuring for Object
const person = { nom:"Ralf", prenom:"SNK" , adress:"rue 123"} ;
// si on veux recuperer les params de l'objet et les assignés dans des variables ;
const nomP = person.nom , prenomP = person.prenom , adressP =person.adress;
console.log("nomP = ",nomP , "prenomP = ", prenomP , "adressP = " ,adressP);
//on va utiliser la fonction de destructuring pour récupérer les valeurs dans les variables
// il faut avoir le même nom des propriétés qu'on veut récupérer depuis l'objet
const {nom ,prenom ,adress} = person ;
console.log("nom = ",nom , "prenom = ", prenom , "adress = " ,adress);
// il est possible de changer le nom des variables
const {nom:nomS ,prenom:prenomS ,adress:adressS} = person ;
console.log("nom = ",nomS , "prenom = ", prenomS , "adress = " ,adressS);
```

Template literals

```
// Template Literals  c'est une fonction étendue de type string
// utilisation de backtick ==> ``  pour creer une string sur plusieurs lignes
// et injecter une variable dans le text par ${nomVariable} sans l'utilisation de la concaténation
const telephone ="XIAOMI REDMI NOTE4" ;
const monTelephone = `Mon telephone est ${telephone}
c'est une marque chinoise
 qui propose des produits de bonne qualité `;
console.log("monTelephone ==> " ,monTelephone)
```

Exercice ES6

```
const exerciceES = `// Exercise 1 - Maybe use an Arrow Function?
var double = function(value) {
    return value * 2;
};
console.log(double(10));

// Exercise 2 - If only we could provide some default values...
var greet = function (name) {
    if (name === undefined) { name = "Max"; }
    console.log("Hello, " + name);
};
greet();
greet("Anna");

// Exercise 3 - Isn't there a shorter way to get all these Values?
var numbers = [-3, 33, 38, 5];
console.log(Math.min.apply(Math, numbers));

// Exercise 4 - I have to think about Exercise 3 ...
var newArray = [55, 20];
Array.prototype.push.apply(newArray, numbers);
console.log(newArray);

// Exercise 5 - That's a well-constructed array.
var testResults = [3.89, 2.99, 1.38];
var result1 = testResults[0];
var result2 = testResults[1];
var result3 = testResults[2];
console.log(result1, result2, result3);

// Exercise 6 - And a well-constructed object!
var scientist = {firstName: "Will", experience: 12};
var firstName = scientist.firstName;
var experience = scientist.experience;
console.log(firstName, experience);`;
const double =(value:number)=>  value*2 ;
console.log(double(10));
const greet = (name:string = 'Ralf'):void=>{
    console.log(`Hello ${name}`);
}
greet() ;
greet("TITI");

const numbersExercice =[-3 , 33 , 38 , 5 ] ;
console.log('Min des valeurs ' , Math.min(...numbersExercice));

const testResults = [3.89 ,2.99,1.38];
const [result1 , result2 , result3] =testResults ;
console.log(result1 , result2 , result3) ;

const scientist = {firstName: "Will", experience: 12};
const {firstName:firstNameExercice , experience :experienceExercie } =scientist ;
console.log(firstNameExercice , experienceExercie );
```

```ts
class Boite{
    public id:number;
    private longeur:number;
    private largeur:number;
    private hauteur:number;
    protected matier:string;
    // par défaut la visiblité de la propriété c'est public
    // par défaut le type de variable c'est any
    color:string;

    // même si on a pas la propriété marque dans la classe
    // la déclaration de la variable avec la portée public ou protected
    // dans la signature de constructor le cree implicitement
    constructor(id:number ,longeur:number , hauteur:number ,matier:string , public marque:string ){
        this.id = id ;
        this.longeur = longeur ;
        this.hauteur = hauteur ;
        this.matier = matier ;
    }
    // par défaut la visibilité des fonction c'est public
    public getVolume(){
        return this.longeur * this.largeur * this.hauteur ;
    }

    private mouvementGauche(){
        console.log("movement vers la gauche boite id: "+this.id);
    }
    private mouvementDroite(){
        console.log("movement vers la à droite boite id: "+this.id);
    }
}

class BoiteFromage extends Boite {
    // le constructeur de la class fille doit faire appel
    // au constructeur de la classe mère
    // il doit passer les valeurs au constructeur de la classe mère si elles sont definis
    constructor(id:number ,longeur:number , hauteur:number ,matier:string , marque:string ){
        super(id , longeur , hauteur, matier , marque);
        // les propriétés et les fonctions qui sont dans la classe mère[boite]
        // sont inaccessible dans la classe fille [BoiteFromage]
        // this.movementDroite() ;
        // this.largeur ;
        this.color="Bleu";
    }
}
let boite1 = new Boite(1 , 10 , 10 , "carton" , "Vache qui rit");
```

```
> boite1
< ▶ Boite {marque: "Vache qui rit", id: 1, longeur: 10, hauteur: 10, matier: "carton"}
>
```

```
class Adresse{
    private _adresse1:string = "no adresse";
    private _adresse2:string;
    private _codePostal:number;
    private _ville:string;
    protected _pays:string;

    constructor(){

    }
    get adresse1(){
        return this._adresse1 ;
    }
    set adresse1(value:string){
        if(value !=null && value.length > 3){
            this._adresse1 = value ;
        }else{
            this._adresse1 = "no adresse"
        }

    }
    get adresse2(){
        return this._adresse2 ;
    }
    set adresse2(value:string){
        this._adresse2 = value;
    }
}
let adresse1 = new Adresse();
console.log("utilisation du get adresse --> ",adresse1.adresse1) ;
console.log("utilisation du set adresse -- ") ;
adresse1.adresse1 = "Drancy" ;
console.log("utilisation du get adresse --> ",adresse1.adresse1) ;
```

```
top  Filter
utilisation du get adresse -->  no adresse
utilisation du set adresse --
utilisation du get adresse -->  Drancy
>
```

Propriété et méthode statique

```
class Helper {
    // propriété static propre à la classe non à l'instance d'objet
    // accessible sans l'instanciation ou la création d'objet
    static PI:number= 3.14 ;
    static calculerDiametre(diametre:number):number{
        return diametre* this.PI ;
    }
}
console.log(2 * Helper.PI);
console.log(Helper.calculerDiametre(2));
```

```
top  Filter
6.28
6.28
>
```

Classe abstract

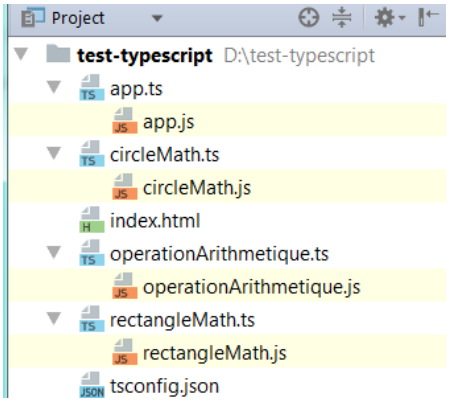
```
// classe abstract
abstract class Moteur{
    // les classes abstract ne peuvent pas être instancié
    // les méthodes abstract doivent être redéfini dans les classes filles qui héritent d'une
classe mère
    // si la classe a une méthode abstract elle doit être déclaré avec le mot clé abstract
    protected _reference: string ;
    protected _nom:string ;
    protected _type: string ;

    constructor(reference:string , nom:string ,type:string){
        this._reference = reference;
        this._nom = nom;
        this._type = type ;
    }
    get reference() {
        return this._reference ;
    }
    set reference(value:string){
        this._reference = value ;
    }
    get nom(){
        return this._nom ;
    }
    set nom(value :string){
        this._nom = value ;
    }
    get type(){
        return this._type;
    }
    set type(value: string){
        this._type = value;
    }
    abstract demarage() :string;
    abstract getInfo():string ;
}
// classe fille --> MoteurDiesel
class MoteurDiesel extends Moteur{
    constructor(reference:string , nom:string ,type:string){
        super(reference,nom, type);
    }
    demarage(): string {
        return "Démarrage Moteur diesel";
    }
    getInfo():string {
        return ` référence => ${this.reference} ,
                nom => ${this.nom}
                type => ${this.type}`;
    }
}
// classe fille --> MoteurElectrique
class MoteurElectrique extends Moteur{
    constructor(reference:string , nom:string ,type:string){
        super(reference,nom, type);
    }
    demarage(): string {
        return "Démarrage Moteur Electrique";
    }
    getInfo():string {
        return ` référence => ${this.reference} ,
                nom => ${this.nom}
                type => ${this.type}`;
    }
}

let moteurAlphaDiesel = new MoteurDiesel("MT-Diesel-001" , "BMW" , "Hybrid");
let moteurAlphaElectrique = new MoteurElectrique("MT-Electric-009" , "TESLA" , "Electric");
// il n'est pas possible de creer une instance à partir d'une classe abstract
// si non on récupère un message : --> Cannot create an instance of the abstract class 'Moteur'
// let moteurPrototype = new Moteur();
```

Namespace

```
// le namespace est un objet {}
namespace FonctionMathematique {
    // la constant PI est défini dans le bloc scope de namespace :FonctionMathematique
    const PI = 3.14 ;
    // export permet d'exposer à l'extérieur la fonction qui est defini dans ce namespace
    // sans le export , la portée de la fonction est défini seulement à l'intérieur de namespace
    export function multiplication(value1 :number , value2:number):number{
        return value1 * value2 ;
    }
    export function addition(value1 :number , value2:number):number{
        return value1 + value2 ;
    }
    export function soustraction(value1 :number , value2:number):number{
        return value2 - value2 ;
    }
    export function cercle(diametre :number ):number{
        return PI*diametre ;
    }
}
// la délation de PI n'impacte pas le bloc scope de namespace :FonctionMathematique
// car elle est défini dans le global scope
const PI = 1.12 ;
console.log('multiplication --> 2 * 4 = ' , FonctionMathematique.multiplication(2,4));
console.log('addition --> 2 + 4 = ' , FonctionMathematique.addition(2,4));
console.log('soustraction --> 4 - 2 = ' , FonctionMathematique.addition(2,4));
// le compilateur détecte que la propriété PI n'est pas défini dans l'objet namespace
:FonctionMathematique
// pour corriger ce problème il faut ajouter le mot clé export sur la const PI
// console.log(' FonctionMathematique.PI =' ,FonctionMathematique.PI) ;
```

Namespaces dans différents fichiers

On va construire une architecture d'un projet on utilisant les namespace .
Les fichiers *.JS sont générés automatiquement par le compilateur
Pour compiler il faut utiliser la commande **tsc** dans l'invite de commande



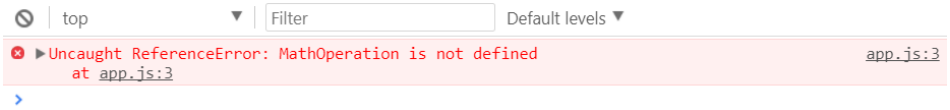

```
namespace MathOperation {
    export function calculeSurface(longeur:number , largeur:number):number{
        return longeur*largeur ;
    }
}
// fichier rectangeMath.ts
```

```
namespace MathOperation {
    const PI :number = 3.14 ;
    export function calculeDiametre (diametre:number): number{
        return diametre * PI ;
    }
}
// fichier circleMath.ts
```

```ts
namespace MathOperation {
    export function multiplication(value1 :number , value2:number):number{
        return value1 * value2 ;
    }
    export function addition(value1 :number , value2:number):number{
        return value1 + value2 ;
    }
    export function soustraction(value1 :number , value2:number):number{
        return value2 - value2 ;
    }
}
// fichier operationArithmetique.ts
```

```ts
// à partir du fichier OperationArithmetique.ts
console.log(MathOperation.addition(2,8));
// à partir du fichier circleMath.ts
console.log(MathOperation.calculeDiametre(2));
// à partir du fichier rectangleMath.ts
console.log(MathOperation.calculeSurface(2,8));

// fichier app.ts
```

```html
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"  content="width=device-width, user-scalable=no, initial-scale=1.0,
maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Learning TypeScript</title>
    <script src="app.js"></script>
</head>
<body>

</body>
</html>
```




```
<!--Au moment de l'exécution le fichier app.js fait appel au namespace défini dans  plusieurs
fichiers-->
<!--la concle nous indique que le namesapce n'est pas defini , car les fichiers-->
<!--circleMath.js , operationArithmetique.js , rectangleMath.js non sont pas inclut-->
 <!--il y a deux solutions :-->
    <!--1 -> inclure les fichiers [*.js] dans la page index.html ou ils sont définis le namespace
utilisé-->
        <!--<script src="circleMath.js"></script>-->
        <!--<script src="operationArithmetique.js"></script>-->
        <!--<script src="rectangleMath.js"></script>-->
    <!--2 -> dans l'invite de commande tsc --outFile app.ts  circleMath.ts operationArithmetique.ts
rectangleMath.ts -->
        <!--cette commande permet de générer un seul fichier app.js qui comporte tout le code-->
        <!--tous le fichiers [.ts] seront mérgés dans un seul ficher .js-->
<!-- fichier index.html -->
```

Les imports de namespace

```
Pour les imports dans typeScripe
/// <reference path ="circleMath.ts" />
/// <reference path ="operationArithmetique.ts" />
/// <reference path ="rectangleMath.ts" />

// a partir du fichier OperationArithmetique.ts
console.log(MathOperation.addition(2,8));
// a partir du fichier circleMath.ts
console.log(MathOperation.calculeDiametre(2));
// a partir du fichier rectangleMath.ts
console.log(MathOperation.calculeSurface(2,8));

// fichier app.ts

// Typescript propose le mot clé [ /// <reference path ="cheminFichier.ts" /> ]
// tsc --outFile app.js circleMath.ts operationArithmetique.ts rectangleMath.ts
// tsc --outFile [fichierEnSortie.JS] et la [liste des fichiers .TS]
```

Quand en compile le fichier app.ts avec la commande **tsc app.ts, typeScript compile le ts en js mais il n'intègre pas les fichiers importés pour que typeScript puisse importer tout le code des autre namespace il faut compiler avec l'option : tsc app.ts -- oneLine app.js**

Plus dans un namesapce

Il est possible de créer des namespace dans un autre namespace

```
circleMath.ts deusieme version
namespace MathOperation {
    const PI :number = 3.14 ;
    export namespace Circle { // --> creation de ns = Circle dans le ns= MathOperation
        export function calculeDiametre (diametre:number): number{
            return diametre * PI ;
        }
    }
}
```

app.ts

```
// on change l'appel pour de la methode calculeDiametre car elle existe dans le ns
MathOperation.Circle
console.log(MathOperation.Circle.calculeDiametre(2));
```

Il est possible de créer un alias pour utiliser le ns s'il est as un chemin trop langue

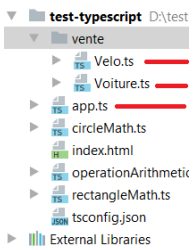
```
import CircleMath = MathOperation.Circle ;
console.log(CircleMath.calculeDiametre(2));
```

Les limites de namespace

Les namespace sont utilisés pour des petits projets, si on travaille sur un grand projet il sera difficile de récupérer toutes les dépendances que le namespace a besoin, c'est pour cette raison typescript propose aussi l'objet module pour structurer les projets

Modules

Utilisation des modules
Création d'un dossier vente avec le module Velo.ts et Voitre.ts



Velo.ts

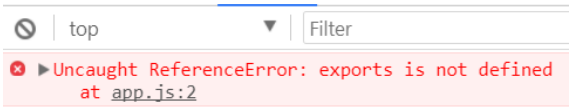
```ts
export const nombreRoue : number = 2 ;
export function afficheVitesse( distance:number , duree :number) :number{
    return distance * duree ;
}
```

Voiture.ts

```ts
export const nombreRoue : number = 4 ;

export function afficheVitesse( distance:number , duree :number) :number{
    return distance * duree ;
}
```

app.ts

```ts
import {nombreRoue , afficheVitesse} from "./vente/Velo";

console.log("nombreRoue =>" ,nombreRoue) ;

// import {les nom de module exposé } from "les fichier sans extention .ts ou tsx ";
si on compile on récurer  un message d'erreur
```



si on compile avec l'option tsc app.ts --outFile app.js
on recuperer une erreur de compilateur Cannot compile modules using opti
cette erreur et du au que javascrit n'a pas ne sais pas le mot clé require("./vente/Velo")

Interface

L'interface c'est un contrat entre celui qui créer un objet un api et celui qui le l'utilise
un exemple

```ts
function viewInfo(person:any){
    console.log(` person.nom = ${person.nom} person.age = ${person.age}`);
}

const person  ={
  nom :"Youssef",
  age :34
};
viewInfo(person);
//person.nom = Youssef person.age = 34
```

On va changer le nom de la propriété de l'objet nom ➔ à nomFamille

```ts
const person  ={
  nomFamille :"Youssef",
  age :34
};
viewInfo(person);
//person.nom = undefined person.age = 34
```

Pour garantir le bon fonctionnement de la méthode il faut utiliser un contrat pour l'utilisation de la méthode

Interface pour une méthode

```
interface NominationPerson {
    nomFamille :string
}
// Controller la signature de la fonction par une interface
function changerNom(person:NominationPerson ){
    console.log(` person.nom = ${person.nomFamille}`);
}

// Controller la signature de la fonction par une interface
function viewInfo(person:NominationPerson){
    console.log(` person.nom = ${person.nomFamille}`);
}

const person  ={
  nomFamille :"Youssef",
  age :34
};
viewInfo(person);
```

Interface et propriétés de l'objet

```
interface NominationPerson {
    nomFamille :string,
    age?:number  // la propriété age est facultatif par le mot clé ? avant la propriété
}
function changerNom(person:NominationPerson ){ //Controller la signature de la fonction par une
interface
    console.log(` person.nom = ${person.nomFamille} `);
}
function viewInfo(person:NominationPerson){  // ON protège la signature de la méthode
    console.log(` person.nom = ${person.nomFamille} `); // par l'objet qui est typé par l'interface
}

const person  ={
  nomFamille :"Youssef",
  age:35
};
viewInfo(person);
viewInfo({nomFamille:"Batman" , age:34});
```

```
interface NominationPerson {
    nomFamille :string;
    age?:number ;  // la propriété age est facultitif
    [propName:string]:number;
}
// Ajouter d'autre propriétés dynamiquement, pour ça il faut avoir les meme type dans les propriétés
déclarées dans l'interface
interface NominationPerson {
    nomFamille :string;
    age?:number ;  // la propriété
    [propName:string]:any;
}
[propName:string] // Ça permet d'accepter d'autre propriétés dans le contrat sans savoir le nom de
la propriété
```

Interface et méthodes

```
interface NominationPerson {
    nomFamille :string;
    age?:number ;  // la propriété age est facultatif
    [propName:string]:any;
    saluer(nom:string) :void ;//définition de la méthode dans saluer qui ne retourne pas de valeur
}
const person :NominationPerson  ={  // c'est une instance avec une référence main pas une classe
  nomFamille :"Youssef",
  age:35 ,
  saluer(nom :string){
      console.log(`Salut Operateur : ${nom}`);
  }
};
// si on définit une nouvelle méthode dans l'interface ,il faut qu'elle soit redéfini ou implémenté
// dans l'objet qui l'utilise
viewInfo(person);

person.saluer("Batman");
```

```
person.nom = Youssef
Salut Operateur : Batman
>
```

Interface et classes

```
interface NominationPerson {
    nomFamille :string;
    age?:number ;  // la propriété age est facultatif
    [propName:string]:any;
    saluer(nom:string) :void ;
}
function viewInfo(person:NominationPerson){
    console.log(` person.nom = ${person.nomFamille} `);
}

class Person implements NominationPerson{
    nomFamille : string;
    saluer(nom:string){
        console.log(`Salut Operateur : ${nom}`);
    }
}

const p1 = new Person();
p1.nomFamille ="Hulk" ;
viewInfo(p1);
p1.saluer("Batman");
// comme dans le monde Java, si on définit une méthode dans l'interface elle doit absolument
// être défini dans la classe qui implémente cette interface, c'est un contrat pour utiliser
 // une implémentation d'une classe ou méthode  , si la propriété ou la méthode est facultatif il
faut dans l'interface spécifié ca par le caractère ? après la propriété ou la méthode
```

Interface et classes

```
interface ICalculerRayon {
    (rayon:number) :number; // on spécifie seulement la déclaration de la fonction
}
let implCalculerRayon :ICalculerRayon;
implCalculerRayon = function(rayon){
    return rayon * 3.14 ;
}
console.log(implCalculerRayon(3));
```

Interface et l'héritage

```
interface INominationPerson {
    nomFamille :string;
    age?:number ;
    [propName:string]:any;
    saluer(nom:string) :void ;
}
interface  IAgePerson  extends INominationPerson{
    age:number  // l'interface IAgePerson hérite de INominationPerson
} // et la propriété Age est obligatoire dans IAgePerson alors que ce n'est pas le cas dans
INominationPerson
const person :IAgePerson  ={  // l'objet person doit avoir l'âge pour respecter le contrat
    nomFamille :"Youssef",
    age:35 ,
    saluer(nom :string){
        console.log(`Salut Operateur : ${nom}`);
    }
};
```

Qu'est ce qui ce passe pour les interfaces après la compilation

```
interface INominationPerson {
    nomFamille :string;
    age?:number ;
    [propName:string]:any;
    saluer(nom:string) :void ;
}
interface  IAgePerson  extends INominationPerson{
    age:number
}
const person :IAgePerson  ={
    nomFamille :"Youssef",
    age:35 ,
    saluer(nom :string){
        console.log(`Salut Operateur : ${nom}`);
    }
};
```

```
"use strict";
var person = {
    nomFamille: "Youssef",
    age: 35,
    saluer: function (nom) {
        console.log("Salut Operateur : " + nom);
    }
};
```

*Dans le fichier app.js , on ne trouve pas la trace des interfaces , car javascript **ne connait pas les interface** , et le compilateur de **Typescript ne crée pas des interfaces** elles sont seulement utilisés au moment de développement*

Generic

```
// Generic
function affiche(donnee:any){
    return donnee ;
}

console.log(affiche("Youssef").length) ;
console.log(affiche(34).length) ;
console.log(affiche({nom:"Youssef" , age:34}).length) ;

// le type any permet d'utiliser le concept de la programmation générique
// car le type any et de type Object il peut prendre n'importe quel type
// le problème c'est que on connait pas le type et la méthode lenght
// n'est pas implémenté dans tous les objet ce que explique les résultats suivants
```

```
top
7
undefined
undefined
>
```

Création d'une fonction générique

```
// Generic
function affiche<T>(donnee:T){
    return donnee ;
}

console.log(affiche("Youssef").length) ;
console.log(affiche(34).length) ;            <-- n'est pas accepté par le compilateur
console.log(affiche<number>("34").length) ;  <--
console.log(affiche<number>(34)) ;
console.log(affiche(34)) ;
console.log(affiche({nom:"Youssef" , age:34}).length) ;  <--
// pour rendre la fonction une fonction générique on ajout <T ou autre lettre>
// le même type T utiliser pour la fonction sera utiliser aussi comme type dans la signature de la méthode
// dans l'exemple typescript détermine le type de variable passé
// et il peut déterminer l'IntelliSense des méthodes supporté par les types des variables
```

Les tableaux et le type générique

```
const ageList : Array<number> = [20,30,40]; // le type Array est par défaut de type générique
// Array<type>
ageList.push(-10);
ageList.push("100"); <-- le compilateur n'accepte pas les string car il sait déjà le type
console.log(ageList);
```

```
▶ (4) [20, 30, 40, -10]
>
```

```
// la fonction a un type générique et elle a une signature avec une liste de type generique
function printAll<T>(args :T[]){
    args.forEach((element) => console.log(element));
}
printAll<string>(["BMW" ,"Mercedes", "Fiat"]) ;
<string>  = pour être explicité
```

Utilisation des types génériques

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

const echo2 :<T> (data: T ) => T = afficherResulat ;
console.log(echo2<string>("Hicho"));
    // en déclare un type => echo2
    // de type générique  => <T>
    // en entrée il récupère de donnée de type => T
    // il retourne un type générique => = T
    // fait Apple à la méthode en sortie => afficherResulat
```

Création des classes génériques

```
class SimpleMath {
    baseValue  ;  // le type par défaut c'est any
    multiplayValue; // le type par défaut c'est any

    calculate(){
        // la valeur retourner est de type any si en ne met pas de type
        return this.baseValue * this.multiplayValue ;
    }
}

const simpleMath = new SimpleMath();
simpleMath.baseValue= 10 ;
simpleMath.multiplayValue = 9 ;
console.log(simpleMath.calculate()) ;

simpleMath.baseValue= "toto" ;
console.log(simpleMath.calculate()) ;
dans l'exemple on a multiplié une chaine de caractère avec un nombre qui donne un résultat
NotANumber est c'est logique
pour rendre cette classe plus générique on va appliquer les changements suivants :
```

```
class SimpleMath <T> {
    baseValue :T ;  // le type par defaut c'est any
    multiplayValue :T; // le type par defaut c'est any

    calculate():number{
        return +this.baseValue * +this.multiplayValue ;
        //le compilateur indique une erreur de type pour les propriétés de la classe
        // il faut les transtipés en type numérique avec , c'est possible de le faire
        // on ajoutant un signe + avant les propriétés
    }
}
// cette solution ne résolut pas le problème il le résolut au moment de compilation
// mais pas au moment de l'exécution
La solution est de limité le type générique par l'héritage

class SimpleMath <T extends number | string> {
    baseValue :T ;
    multiplayValue :T;

    calculate():number{
        return +this.baseValue * +this.multiplayValue ;
    }
}

const simpleMath = new SimpleMath<number>();
// le problème sera détecté au moment de la compilation et non plus au moment de l'exécution
simpleMath.baseValue= "toto" ; // pour la valeur "toto"
simpleMath.baseValue = 10 ; //
simpleMath.multiplayValue = 9 ;
console.log(simpleMath.calculate()) ;

//une autre solution
const simpleMath2  = new SimpleMath<string>();
simpleMath2.baseValue = "10" ; // les nombres déclarés comme des String seront transtipé en
nombre par typescript
simpleMath2.multiplayValue = "9" ;
console.log(simpleMath.calculate()) ;
```

Petite config dans tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false
  },
  "exclude": [
    "node_modules"
  ]
}
```

Utiliser plusieurs types génériques

```
class SimpleMath <W extends U ,U extends number | string , T extends number | string> {
    baseValue :T ;
    multiplayValue :U;
    xVal : W;

    calculate():number{
        return +this.baseValue * +this.multiplayValue * +this.xVal  ;
    }
}

const simpleMath = new SimpleMath<string, string, number>();
simpleMath.baseValue= 10 ;
simpleMath.multiplayValue = "20" ; //
simpleMath.xVal = "9" ;
console.log(simpleMath.calculate()) ;
```

Decorator

Création de Decorator des classes

Decorateur de classe c'est une fonction qui accepte une fonction constructeur dans sa signature et retourne , soit undefined, soit la fonction constructeur fourni .Retourner un undefined est équivalent à la fonction constructeur fourni .La clase décorateur est utilisé pour modifier le constructeur de la classe d'une maniére ou d'une autre .Si la classe decorator retourne undefined ,le constructeur d'origine reste le même.si le decorateur retourne quelque chose , la valeur retourné va etre utiliser pour redéfinir(override) le constructeur de la classe origine.

Typescritp celui qui gère l'attachement de la fonction décorateur à la classer par @nomFn
Récupère la fonction constructeur de la classe et le passe à la fonction décorateur

```
function logger(constructorFn: Function){
    console.log("logger =====>");
    console.log(constructorFn);
}

@logger
class Person {
    constructor(){
        console.log("constructor ==> Person");
    }
}
```

Le constructeur Injecté par typescripte

```
logger =====>
ƒ Person() {
        console.log("constructor ==> Person");
    }
>
```

Decorator Factory

Decorator factory c'est une fonction qui accepte n'importe quel nombre d'arguments et elle doit retourner un des types des fonctions decortor ai dessous
Comme dans angular nous consommant des decorators qui sont déjà défini et nous n'avant pas de les implémenter .exp : @view

```
Dans le fichier tsconfig.js il faut activer
l'option "experimentalDecorators": true
Pour utiliser les decorators
```

```
function logged(constructorFn: Function){
    console.log(constructorFn);
}

@logged
class Person {
    constructor(){
        console.log("constructor ==> Person");
    }
}

// Factoty
function logging(value:boolean):any{
    return value ? logged : null ;
}

@logging(true)
class Car {

}
```

Le pattern factory permet de décider quel objet faut-il créer , ici il est implémenté pour soit logger ou pas la classe Car
La classe est attaché à la fonction logging par le resultat de la factory

```
logger =====>
ƒ Person() {
        console.log("constructor ==> Person");
    }
logger =====>
ƒ Car() {
    }
```

@logging(true)

```
logger =====>
ƒ Person() {
        console.log("constructor ==> Person");
    }
```

@logging(false)

Useful Decorator

La function paint() n'été pas define dans la classe Plant, on va greffer ce comportement par le decorator printable
Le decorator va override ou redefinir le constructeur de la classe Plant , on ajoutant dans le prototype de la classe la méthode print()

```
function printable(constructorFn:Function){
    constructorFn.prototype.print = function(){
        console.log(this);
    }
}
@printable
class Plant{
    name ="Fleur rouge";
}
const plant = new Plant();
(<any>plant).print();
```

```
▼ Plant {name: "Fleur rouge"}
    name: "Fleur rouge"
  ▶ __proto__: Object
```

Typescript passe le constructeur en paramètre et la méthode surcharge le constructeur par la méthode print et sera utilisé au moment de l'exécution pas au moment de la compilation

On caste l'objet à any car typescript est incapable de savoir la méthode print dans cet objet , c'est un petit bug

Utiliser plusieurs decorators

```
@logging(true)       // decorator @logging
@printable           // decorator @printable
class Plant{
    name ="Fleur rouge";
}
const plant = new Plant();
(<any>plant).print();
```

```
logger =====>
ƒ Plant() {
        this.name = "Fleur rouge";
    }
▶ Plant {name: "Fleur rouge"}
```

Decorator pour une méthode

La fonction pour décorer une méthode est une fonction qui accepte trois paramètres (l'objet qui possède la propriété, la clé de la propriété, [une chaine ou symbole] , et un descripteur de propriété)

La fonction pour décorer une méthode c'est une factory

```
class Maison{
    typeMaison:string;
    constructor(type:string){
        this.typeMaison = type ;
    }

    build(){
        console.log("contruction encours")
    }
}

const maison = new Maison("Appat T3");
maison.build();

maison.build= function () {
    console.log("demolition") ;
}
maison.build();
```

```
contruction encours
demolition
```

```
function editable(value:boolean){
    return function(target :any , propName :string , descriptor:PropertyDescriptor){
        // PropertyDescriptor from ==> lib.d.ts
        descriptor.writable = value;
    }
}

class Maison{
    typeMaison:string;
    constructor(type:string){
        this.typeMaison = type ;
    }

    @editable(false)
    build(){
        console.log("contruction encours")
    }
}

const maison = new Maison("Appat T3");
maison.build();

maison.build= function () {
    console.log("demolition") ;
}
maison.build();
```

Typescript set d'après le contexte les différentes valeurs de la fonction

On a pas d'erreurs dans la compilation mais au moment de l'exécution , décorateur editable va interdire la redéfinition de la méthode

```
▶Uncaught TypeError: Cannot assign to read only property 'build' of object '#<Maison>'
```

Decorator pour les paramètres des méthodes

Le decorator pour un paramètre une fonction qui accepte trois paramètres (l'objet qui possède la méthode qui contient le paramètre décoré, la clé de propriété de la propriété [ou indéfinie pour un paramètre du constructeur], et l'ordinal index du paramètre)

La valeur de retour de ce décorateur est ignorée

Ce décorateur n'est pas un factory mais un décorateur

```
function printInfo(target:any , methodName:string ,paramIndex:number){
    console.log("target =>" ,target);
    console.log("methodName =>" ,methodName);
    console.log("paramIndex =>", paramIndex);
}

class Course{
    name:string;
    constructor(name:string){
        this.name = name ;
    }
    printStudentNumbers(node:string ,@printInfo printAll:boolean){
        if(printAll){
            console.log("2000");
        }else{
            console.log("500");
        }
    }
}
```

```
target => ▼{printStudentNumbers: ƒ, constructor: ƒ}
            ▶ printStudentNumbers: ƒ (node, printAll)
            ▶ constructor: ƒ Course(name)
            ▶ __proto__: Object
methodName => printStudentNumbers
paramIndex => 1
```

1 = 2eme param de la méthode , index commence par 0

Decorator pour les property

???