TypeScript

1. **Introduction**
2. **Définition**

TypeScript est un langage de programmation open-source développé par Microsoft qui étend JavaScript en ajoutant des fonctionnalités de typage statique. Contrairement à JavaScript, qui est un langage à typage dynamique, TypeScript permet aux développeurs de spécifier des types de variables, des paramètres de fonction, des valeurs de retour et bien plus encore. Ces types permettent d'effectuer une vérification statique des erreurs pendant la phase de développement, améliorant ainsi la robustesse du code.

En résumé, TypeScript est une surcouche de typage statique pour JavaScript, permettant aux développeurs de détecter et de corriger des erreurs potentielles avant l'exécution du code, offrant ainsi une meilleure expérience de développement, une maintenance simplifiée et une meilleure collaboration au sein des équipes de développement. Il se compile en JavaScript standard et est compatible avec les navigateurs modernes et les environnements d'exécution Node.js.

1. **Environnement setup**

* **Install nodeJs**
* **Install typescript (npm i -g typescript)**
* **Voir la version (npm tsc -v)**
* **Creation d’un fichier ts (npx tsc main.ts)**
* **Recompiler le fichier pour n’importe quelle changement(npx tsc main --watch)**

1. **Types**
2. **Variable Types**

Boolean Number String 🡺 basic type

Null Undefined 🡺 Null et undefined don’t des types et des subtypes

let isBeginner: boolean = true;

let tota: number = 0;

let name: string = "Vishwas";

1. **Null et undefined**

let n: null = null;

let u: undefined;

undefined et null sont classifié comme subtype de tous les types

let isNewx: number = null;

let muName: string = undefined;

1. **Array**

let List1: number[] = [1, 2, 3];

let list2: Array<number> = [1, 2, 3];

Un tableau de types mixtes est possible, mais : (le nombre d'éléments est fixe et l'ordre des valeurs doit correspondre à l'ordre des types)

let person1: [string, number] = ["Chris", 22];

1. **Enum**

En TypeScript, une énumération (**enum**) est une fonctionnalité qui permet de définir un ensemble de valeurs constantes nommées. Cela permet de créer un type spécial qui peut prendre l'une des valeurs spécifiées dans l'énumération.

Voici un exemple pour mieux comprendre les énumérations :

enum Couleur {

  Rouge,

  Vert,

  Bleu,

  Jaune

}

let couleurPreferee: Couleur = Couleur.Vert;

console.log(couleurPreferee); // Affiche : 1 (l'index de la valeur 'Vert' dans l'énumération)

// Vous pouvez également accéder aux valeurs individuelles de l'énumération par leur nom

let couleurAutre: Couleur = Couleur.Rouge;

console.log(couleurAutre); // Affiche : 0 (l'index de la valeur 'Rouge' dans l'énumération)

tout à fait ! Les énumérations (**enum**) en TypeScript sont souvent utilisées pour déclarer des constantes.

enum Couleurs {

  Rouge = "#FF0000",

  Vert = "#00FF00",

  Bleu = "#0000FF",

}

const couleurPreferee: Couleurs = Couleurs.Vert;

console.log(couleurPreferee); // Affiche "#00FF00"

1. **Any**

Le type **any** en TypeScript est utilisé pour indiquer qu'une variable peut avoir n'importe quel type de valeur. Cela permet de désactiver la vérification de type statique pour cette variable, ce qui signifie que TypeScript ne vérifiera pas le type de la valeur affectée à une variable de type **any**

let randomValue: any = 10;

randomValue = true;

randomValue = "Houssam";

//AnyTyoe don't verify anithing

console.log(randomValue.name);

randomValue();

randomValue.toUpperCase();

1. **Unknown Type**

Le type **unknown** en TypeScript est utilisé pour représenter une valeur dont le type est inconnu à la compilation. C'est une manière plus sûre d'utiliser des valeurs dont le type n'est pas clair ou certain, par rapport au type **any**. Contrairement à **any**, **unknown** impose des contraintes plus strictes lors de l'utilisation de la valeur, ce qui améliore la sécurité du typage.

Lorsqu'une variable est de type **unknown**, TypeScript exige que vous effectuiez une vérification de type avant d'accéder ou d'utiliser cette variable. Cela signifie que vous devez d'abord vérifier son type avant de pouvoir effectuer des opérations spécifiques avec cette valeur.

function afficherLongueur(valeur: unknown) {

  if (typeof valeur === 'string') {

    console.log(`La longueur de la chaîne est : ${valeur.length}`);

  } else {

    console.log('Type de valeur inconnu.');

  }

}

let v1: unknown = "Bonjour";

let v2: unknown = 42;

afficherLongueur(v1); // Affiche "La longueur de la chaîne est : 7"

afficherLongueur(v2); // Affiche "Type de valeur inconnu."

let randomValue1: unknown = 10;

randomValue1 = true;

randomValue1 = "Houssam";

function hasName(obj: any): obj is { name: string } {

  return !!obj && typeof obj === "object" && "name" in obj;

}

if (hasName(randomValue1)) {

  console.log(randomValue1.name);

}

// we nedd to use a type assertion to convert the type sysytem that we know

(randomValue1 as string).toUpperCase();

1. **Type inference**

let a

a=10

a=true

let b=20

// b=true error because b is a number that is a type inference

1. **Union Type**

let multiType : number | boolean

multiType=20

multiType=true

1. **Function Type et optionnel type**

function add1(num1:number,num2?:number):number{

    if(!num2){

        return num1

    }

    else{

    return num1+num2

    }

}

add1(5)

1. **Interface**

En TypeScript, une "interface" est un moyen de définir une structure de type pour les objets. Elle permet de définir un contrat en spécifiant les propriétés et les types de données attendus pour un objet. Les interfaces sont utilisées pour décrire la forme des objets et fournissent une vérification statique des types lors de la compilation.

interface Point {

    x: number;

    y: number;

}

interface SearchFunc {

    (source: string, subString: string): boolean;

}

// In this example, the SearchFunc interface defines a single method,

interface Point {

    x: number;

    y: number;

}

class MyPoint implements Point {

    x: number;

    y: number;

    constructor(x: number, y: number) {

        this.x = x;

        this.y = y;

    }

}

// In this example, MyPoint is a class that implements the Point interface. It must have two properties x and y of type number.

interface Person{

      firstName:string

      lastName:string

      age?:number//optional

}

function fullName(person: Person){

    console.log(`${person.firstName} ${person.lastName}`)

}

let p={

    firstName:'Bruce',

    lastName:'wayne',

    test:'test'

}

fullName(p)

* **Difference entre Type et interface**

**interface** :

* Une interface est spécifiquement utilisée pour définir la structure de types pour les objets et les classes.
* Elle peut être étendue, c'est-à-dire qu'une interface peut hériter d'une ou plusieurs autres interfaces.
* Les interfaces peuvent être implémentées par des classes.
* Vous pouvez définir des propriétés obligatoires et des propriétés optionnelles en utilisant le symbole **?**.
* Lorsqu'une interface est étendue, toutes les propriétés de l'interface de base sont incluses dans la nouvelle interface.
* Les interfaces sont plus adaptées pour décrire la forme d'un objet ou pour créer des contrats entre différentes parties du code.

interface Person {

  firstName: string;

  lastName: string;

  age: number;

  email?: string;

}

**type** :

* Un type peut être utilisé pour définir n'importe quel type de données, pas seulement des objets.
* Il permet de créer des alias pour les types existants ou de créer des types basés sur des unions, des intersections, etc.
* Les types peuvent être étendus, mais cela se fait en utilisant **&** pour les intersections ou **|** pour les unions.
* Les types ne peuvent pas être implémentés par des classes.
* Ils sont plus flexibles et polyvalents que les interfaces, car ils permettent de manipuler des types de données de manière plus complexe.

type Fruit = "Apple" | "Orange" | "Banana";

type Person = {

  firstName: string;

  lastName: string;

  age: number;

  email?: string;

};

En général, utilisez des **interfaces** lorsque vous travaillez avec des objets et des classes, et utilisez des **types** pour créer des alias ou définir des types plus complexes, notamment des unions ou des intersections de types. Dans certains cas, la décision entre **interface** et **type** peut dépendre de préférences personnelles ou de conventions de code dans un projet donné. L'important est de rester cohérent dans l'utilisation des deux afin de rendre votre code clair et facilement compréhensible.

1. **Acces modifiers in classes**

En TypeScript, les "access modifiers" sont utilisés pour contrôler l'accès aux membres (propriétés et méthodes) d'une classe. Ils permettent de spécifier quelles parties du code peuvent accéder aux membres de la classe et comment ils peuvent être manipulés. Il existe trois access modifiers en TypeScript :

1. **public** :
   * C'est le modificateur par défaut si aucun access modifier n'est spécifié.
   * Les membres marqués comme **public** sont accessibles depuis n'importe quelle partie du code, y compris en dehors de la classe.
2. **private** :
   * Les membres marqués comme **private** ne sont accessibles que depuis l'intérieur de la classe où ils ont été déclarés.
   * Ils ne sont pas accessibles depuis l'extérieur de la classe, y compris depuis les sous-classes.
3. **protected** :
   * Les membres marqués comme **protected** sont accessibles depuis la classe où ils ont été déclarés, ainsi que depuis les sous-classes de cette classe.
   * Ils ne sont pas accessibles depuis l'extérieur de la classe ni depuis les instances de la classe.

Voici un exemple de l'utilisation des access modifiers en TypeScript :

class Person {

  public name: string; // public par défaut

  private age: number;

  protected email: string;

  constructor(name: string, age: number, email: string) {

    this.name = name;

    this.age = age;

    this.email = email;

  }

  public sayHello() {

    console.log(`Hello, my name is ${this.name}.`);

    this.secretMethod(); // OK, la méthode peut accéder aux membres privés et protégés de la classe

  }

  private secretMethod() {

    console.log("This is a secret method.");

  }

}

const john = new Person("John", 30, "john@example.com");

console.log(john.name); // OK, accès au membre public "name"

console.log(john.age); // Erreur, "age" est privé et inaccessible depuis l'extérieur de la classe

console.log(john.email); // Erreur, "email" est protégé et inaccessible depuis l'extérieur de la classe

john.sayHello(); // OK, méthode publique "sayHello" peut être appelée

john.secretMethod(); // Erreur, "secretMethod" est privée et inaccessible depuis l'extérieur de la classe

Dans cet exemple, nous avons une classe **Person** avec trois membres : **name**, **age**, et **email**, chacun étant marqué avec un access modifier différent. Vous pouvez voir comment chaque access modifier affecte l'accès aux membres de la classe et aux méthodes à l'intérieur et à l'extérieur de la classe.

Les access modifiers en TypeScript vous permettent de contrôler la visibilité des membres d'une classe et d'encapsuler correctement le code pour améliorer la sécurité et la lisibilité de votre application.

1. **Omit**

**Omit** est un type utilitaire en TypeScript qui permet de créer un nouveau type en excluant certaines propriétés spécifiées d'un type existant. Cela vous permet de créer une version modifiée du type original avec certaines propriétés omises.

type TypeOriginal = {

  nom: string;

  age: number;

  email: string;

  adresse: string;

};

// Exclure les propriétés 'age' et 'email' du TypeOriginal

type NouveauType = Omit<TypeOriginal, 'age' | 'email'>;

// NouveauType aura la forme suivante :

// type NouveauType = {

//   nom: string;

//   adresse: string;

// }