

# TypeScript

M. Serigne Diagne

Ingénieure étude et développement

sdiagne148@gmail.com

## Introduction à TypeScript

TypeScript est une extension de JavaScript qui ajoute une syntaxe pour les types. Il s'agit d'un **superset syntaxique** de JavaScript, ce qui signifie qu'il reprend la syntaxe de base de JavaScript tout en y ajoutant des fonctionnalités, notamment le typage statique.

### Pourquoi utiliser TypeScript ?

- **Typage statique** : Permet de définir les types de données dans le code, ce qui améliore la lisibilité et réduit les erreurs.
- **Vérification au moment de la compilation** : TypeScript détecte les erreurs de type avant l'exécution, contrairement à JavaScript qui les détecte en cours d'exécution.
- **Documentation implicite** : Les types explicitement définis éliminent le besoin de deviner ou de consulter la documentation pour comprendre le type des données manipulées.

**Exemple** : TypeScript signalera une erreur si une chaîne est passée à une fonction qui attend un nombre, alors que JavaScript l'acceptera sans alerte.

### Comment utiliser TypeScript ?

- Utilisez le **compilateur officiel TypeScript** pour convertir le code TypeScript en JavaScript.
- **Éditeurs supportés** : Certains éditeurs comme Visual Studio Code intègrent TypeScript et affichent les erreurs directement pendant la saisie.

## Le Compilateur TypeScript

TypeScript est **transpilé en JavaScript** à l'aide d'un compilateur, ce qui permet de l'exécuter partout où JavaScript fonctionne.

### Installation du compilateur

Pour installer le compilateur officiel TypeScript dans un projet npm, exécutez :

---

```
npm install typescript --save-dev
```

---

Cela ajoute TypeScript comme dépendance de développement. Vous pouvez ensuite exécuter le compilateur avec :

---

*npx tsc*

---

Cela affiche la version du compilateur et les commandes disponibles.

### Configuration du compilateur

Le compilateur peut être configuré via un fichier **tsconfig.json**, que vous pouvez créer avec :

---

*npx tsc --init*

---

Cela génère un fichier avec des options par défaut, comme :

- **target** : es2016
- **module** : commonjs
- **strict** : true

Exemple d'ajout personnalisé :

```
ts tsconfig.json > ...
1  {
2    "include": ["src"],
3    "compilerOptions": {
4      "outDir": "./build"
5    }
6 }
```

Cela indique au compilateur de convertir les fichiers TypeScript du dossier **src/** en JavaScript dans le dossier **build/**.

**NB** : TypeScript peut être intégré dans divers projets à l'aide de modèles comme **create-react-app**, un **projet Node.js** de démarrage, ou des outils comme **webpack**.

## Types Simples en TypeScript

TypeScript prend en charge les types primitifs suivants :

### Principaux types primitifs :

- **boolean** : valeurs true ou false.
- **number** : nombres entiers et décimaux.

- **string** : chaînes de caractères, ex. "TypeScript Rocks".

### Types primitifs moins courants :

- **bigint** : pour des nombres entiers très grands ou très petits.
- **symbol** : pour créer des identifiants globalement uniques.

## Attribution de Type

Deux manières principales d'attribuer un type à une variable :

### 1. Type explicite

On précise explicitement le type :

```
1 let firstName: string = "Serigne";
2
```

Cela améliore la lisibilité et l'intentionnalité.

### 2. Type implicite

TypeScript infère le type selon la valeur assignée :

```
let firstName = "Serigne";
```

Cette méthode est rapide et utile pour le développement et les tests.

## Erreurs d'attribution de type

TypeScript génère une erreur si les types ne correspondent pas :

```
> TS types.ts > ...
1 let firstName: string = "Serigne";
2 firstName = 33; // Erreur : le type attendu est une chaîne, pas un nombre.
```

Même avec une attribution implicite, TypeScript détecte les erreurs de type.

## Problèmes d'inférence

Dans certains cas, TypeScript ne peut pas inférer correctement le type, il utilise alors le type **any**, désactivant ainsi la vérification de type :

```
const json = JSON.parse("55");
console.log(typeof json); // Peut être n'importe quel type.
```

Pour éviter cela, activez **noImplicitAny** dans le fichier **tsconfig.json**.

## Note sur les types primitifs capitalisés :

Utilisez toujours les valeurs en minuscules (boolean, number, etc.) et non les versions en majuscules (Boolean), sauf dans des cas très spécifiques.

## Types Spéciaux en TypeScript

TypeScript propose des types spécifiques qui ne représentent pas nécessairement des données classiques.

### Type : any

- **Description** : Désactive la vérification des types, permettant d'utiliser n'importe quel type de donnée.
- **Exemple** :

```
let v: any = true;
v = "string"; // Aucun problème
Math.round(v); // Pas d'erreur
```

- **Inconvénient** : Pas de sécurité de type ni de complétion automatique. À éviter autant que possible.

### Type : unknown

- **Description** : Alternative plus sûre à any, empêchant l'utilisation directe des données sans vérification ou conversion de type.
- **Exemple** :

```
let w: unknown = 1;
w = "string"; // Pas d'erreur
if (typeof w === 'object' && w !== null) {
  (w as { runANonExistentMethod: Function }).runANonExistentMethod()
```

- **Utilisation** : Idéal lorsque le type est inconnu initialement. Pour utiliser la donnée, il faut effectuer une **conversion de type** avec **as**.

### Type : never

- **Description** : Utilisé pour représenter des situations où une valeur ne peut jamais exister.
- **Exemple**

```
let x: never = true; // Erreur : Type 'boolean' n'est pas assignable à 'never'.
```

- **Utilisation** : Rare, principalement dans des cas avancés comme les **génériques**.

### Types : undefined & null

- **Description** : Représentent respectivement les primitives JavaScript undefined et null.
- **Exemple**

```
let y: undefined = undefined;
let z: null = null;
```

- **Note** : Leur utilité augmente si **strictNullChecks** est activé dans tsconfig.json

## Les Tableaux en TypeScript

TypeScript fournit une syntaxe spécifique pour typer les tableaux.

### Déclaration de type pour les tableaux

Un tableau peut être typé pour contenir uniquement un type particulier :

```
const names: string[] = [];
names.push("Dylan"); // Pas d'erreur
// names.push(3); // Erreur : 'number' n'est pas assignable à 'string'.
```

### Tableaux en lecture seule (readonly)

Avec le mot-clé **readonly**, un tableau devient immuable :

```
const names1: readonly string[] = ["Dylan"];
names1.push("Jack"); // Erreur : 'push' n'existe pas sur 'readonly string[]'.
```

### Inférence de type

TypeScript peut déduire automatiquement le type d'un tableau selon ses valeurs initiales :

```
const numbers = [1, 2, 3]; // Inféré comme 'number[]'
numbers.push(4); // Pas d'erreur
numbers.push("2"); // Erreur : 'string' n'est pas assignable à 'number'.

let head: number = numbers[0]; // Pas d'erreur
```

## Tuples en TypeScript

Un tuple est un tableau typé avec une longueur et des types prédéfinis pour chaque index.

### Définir un tuple

Pour définir un tuple, on spécifie le type de chaque élément :

```
let ourTuple: [number, boolean, string];
ourTuple = [5, false, 'Coding God was here']; // Correct
ourTuple = [false, 'Wrong Order', 5]; // Erreur : l'ordre des types est important
```

### Tuples en lecture seule (readonly)

Pour éviter toute modification accidentelle, on peut utiliser le mot-clé readonly :

```
const our_READONLYTuple: readonly [number, boolean, string] = [5, true, 'The Real Coding God'];
our_READONLYTuple.push('Error'); // Erreur : impossible de modifier un tuple readonly
```

### Tuples nommés

Les tuples nommés offrent une meilleure lisibilité en ajoutant des étiquettes aux index :

```
const graph: [x: number, y: number] = [55.2, 41.3];
```

### Destructuration de tuples

Comme les tuples sont des tableaux, ils peuvent être déstructurés :

```
const graph1: [number, number] = [55.2, 41.3];
const [x1, y1] = graph1;
```

## Les Types d'Objets en TypeScript

TypeScript propose une syntaxe spécifique pour typer les objets.

### Déclaration d'un Objet Typé

Chaque propriété de l'objet doit respecter le type défini :

```
const car: { type: string, model: string, year: number } = {  
    type: "Toyota",  
    model: "Yaris",  
    year: 2010  
};
```

Les types d'objets peuvent être définis séparément et réutilisés.

## Inférence des Types

TypeScript peut deviner les types des propriétés à partir des valeurs assignées :

```
const car1 = { type: "Toyota" };  
car1.type = "Ford"; // Aucun problème  
car1.type = 2; // Erreur : Type 'number' non assignable à 'string'
```

## Propriétés Optionnelles

Les propriétés peuvent être rendues optionnelles avec `?` :

```
const car2: { type: string, mileage?: number } = { type: "Toyota" }; // Pas d'erreur  
car2.mileage = 2000; // Ajout possible
```

## Signatures d'Index

Elles permettent de définir des objets avec des propriétés dynamiques :

```
const nameAgeMap: { [index: string]: number } = {};  
nameAgeMap['Jack'] = 25; // Pas d'erreur  
nameAgeMap['Mark'] = "Fifty"; // Erreur : Type 'string' non assignable à 'number'
```

**Astuce** : Les signatures d'index peuvent être exprimées avec des types utilitaires comme `Record<string, number>`.

## Les Enums en TypeScript

Un **enum** est une structure spéciale qui représente un groupe de constantes (valeurs immuables). TypeScript propose deux types principaux : les enums numériques et les enums de chaînes de caractères.

### Enums Numériques (Par Défaut)

Par défaut, les enums numériques débutent à 0 et incrémentent automatiquement :

```
enum CardinalDirections {
    North, // 0
    East, // 1
    South, // 2
    West // 3
}
console.log(CardinalDirections.North); // Affiche : 0
```

⚠ Une valeur non définie dans l'enum génère une erreur :

```
currentDirection = 'North'; // Erreur : non assignable à 'CardinalDirections'
```

### Enums Numériques (Initialisés)

Le premier élément peut être initialisé, les suivants sont incrémentés automatiquement :

```
enum CardinalDirections1 {
    North = 1, // 1
    East, // 2
    South, // 3
    West // 4
}
console.log(CardinalDirections1.West); // Affiche : 4
```

### Enums Numériques (Complètement Initialisés)

Chaque valeur peut être explicitement définie :

```
enum StatusCodes {
    NotFound = 404,
    Success = 200,
    Accepted = 202,
    BadRequest = 400
}
console.log(StatusCodes.NotFound); // Affiche : 404
```

### Enums de Chaînes de Caractères

Les enums peuvent également contenir des chaînes, ce qui améliore leur lisibilité :

```
enum CardinalDirections2 {
    North = 'North',
    East = 'East',
    South = 'South',
    West = 'West'
}
console.log(CardinalDirections2.North); // Affiche : "North"
```

### Bonnes Pratiques :

Bien que TypeScript permette de mélanger les valeurs numériques et de chaînes dans un enum, il est déconseillé de le faire pour maintenir la clarté et la cohérence.

## Type Alias et Interfaces en TypeScript

TypeScript permet de définir des types séparément des variables qui les utilisent. **Les Type Alias et les Interfaces** facilitent le partage et la réutilisation des types dans différents contextes.

### Type Alias

Un Type Alias est une manière de donner un nom personnalisé à un type, qu'il soit primitif ou complexe :

```
type CarYear = number;
type CarType = string;
type CarModel = string;

type Car3 = {
    year: CarYear;
    type: CarType;
    model: CarModel;
};

const car3: Car3 = {
    year: 2001,
    type: "Toyota",
    model: "Corolla",
};
```

### Interfaces

Les interfaces ressemblent aux Type Alias, mais elles ne s'appliquent qu'aux types d'objets :

```
interface Rectangle {
    height: number;
    width: number;
}

const rectangle: Rectangle = {
    height: 20,
    width: 10,
};
```

### Extension des Interfaces

Une interface peut être étendue pour inclure de nouvelles propriétés tout en conservant celles d'une interface existante :

```
interface ColoredRectangle extends Rectangle {
    color: string;
}
const coloredRectangle: ColoredRectangle = {
    height: 20,
    width: 10,
    color: "red",
};
```

## Différences Principales

- **Type Alias** : Peuvent définir n'importe quel type (primitif, objet, union, etc.).
- **Interfaces** : Se limitent aux types d'objets mais sont extensibles et plus adaptées aux structures complexes.

Ces deux approches sont interchangeables dans certains cas, mais les **interfaces** sont souvent privilégiées pour modéliser des objets.

## Union Types en TypeScript

Les **Union Types** permettent de définir une variable ou un paramètre pouvant prendre plusieurs types différents, comme un mélange de string et number.

### Définir une Union avec | (OR)

On utilise le symbole | pour indiquer qu'une valeur peut être de plusieurs types :

```
function printStatusCode(code: string | number) {
    console.log(`My status code is ${code}.`);
}

printStatusCode(404);      // OK : number
printStatusCode('404');    // OK : string
```

### Problèmes avec les Union Types

Il est important de connaître le type exact avant d'utiliser des méthodes spécifiques, pour éviter des erreurs :

```
function printStatusCode1(code: string | number) {
    console.log(code.toUpperCase());
    // Erreur : La méthode 'toUpperCase()' n'existe pas pour le type 'number'.
}
```

### Solution : Vérification de type (typeof)

Pour éviter ces erreurs, vérifiez le type avant d'appeler une méthode spécifique :

```
function printStatusCode2(code: string | number) {
  if (typeof code === 'string') {
    console.log(code.toUpperCase()); // OK : code est une chaîne
  } else {
    console.log(code); // OK : code est un nombre
  }
}
```

## Les Fonctions en TypeScript

TypeScript offre une syntaxe spécifique pour typifier les paramètres et les valeurs de retour des fonctions.

### Retour de Fonction

Le type de la valeur retournée peut être explicitement défini. Si aucun type n'est spécifié, TypeScript essaiera de l'inférer.

```
function getTime(): number {
  return new Date().getTime(); // Retourne un nombre
}
```

### Retour de type void

Le type void indique qu'une fonction ne retourne aucune valeur.

```
function printHello(): void {
  console.log('Hello!');
}
```

### Paramètres de Fonction

Les paramètres des fonctions sont typés de manière similaire aux déclarations de variables.

```
function multiply(a: number, b: number): number {
  return a * b;
}
```

### Paramètres Optionnels

Les paramètres peuvent être marqués comme optionnels avec le ?

```
function add(a: number, b: number, c?: number): number {
  return a + b + (c || 0);
}
```

### Paramètres par Défaut

Pour les paramètres avec des valeurs par défaut, la valeur par défaut vient après l'annotation de type.

```
function pow(value: number, exponent: number = 10): number {
  return value ** exponent;
}
```

## Paramètres Nommés

Les paramètres peuvent aussi être typés avec des objets nommés.

```
function divide({ dividend, divisor }: { dividend: number, divisor: number }): number {
  return dividend / divisor;
}
```

## Paramètres Rest

Les paramètres rest sont toujours des tableaux et peuvent être typés comme tels.

```
function add1(a: number, b: number, ...rest: number[]): number {
  return a + b + rest.reduce((p, c) => p + c, 0);
}
```

## Alias de Type pour Fonctions

Les types de fonctions peuvent être définis avec des alias.

```
type Negate = (value: number) => number;

const negateFunction: Negate = (value) => value * -1;
```

## Résumé

- **Types de retour** : définissez explicitement le type de retour ou laissez TypeScript l'inférer.
- **Paramètres** : spécifiez les types des paramètres, marquez-les comme optionnels, ou utilisez des paramètres par défaut et rest.
- **Alias de type** : utilisez des alias pour définir des types de fonctions réutilisables.

## Casting en TypeScript

Le **casting** est le processus qui consiste à forcer un type pour une variable, souvent utilisé pour résoudre des erreurs de type ou lorsqu'une bibliothèque fournit un type incorrect.

### Casting avec as

Utilisez le mot-clé **as** pour forcer un type sur une variable.

```
💡
let x2: unknown = 'hello';
console.log((x as string).length); // Affiche la longueur de la chaîne
```

Cependant, le casting ne change pas les données internes de la variable. Par exemple :

```
💡  
let x2: unknown = 'hello';  
console.log((x as string).length); // Affiche la longueur de la chaîne
```

TypeScript vérifiera également si le cast est logique et générera une erreur s'il semble incorrect :

```
console.log((4 as string).length); // Erreur : Conversion du type 'number' en 'string' est probablement une erreur
```

### Casting avec <>

Une autre syntaxe pourcaster est d'utiliser `<>`, mais cette méthode ne fonctionne pas avec TSX (React).

```
let x4: unknown = 'hello';  
console.log(<string>x.length); // Affiche la longueur de la chaîne
```

### Force Casting

Pour contourner les erreurs de type, vous pouvez d'abordcaster la variable en `unknown`, puis dans le type cible.

```
💡  
let x5 = 'hello';  
console.log(((x5 as unknown) as number).length); // Retourne undefined car x n'est pas réellement un nombre
```

## Résumé

- **Casting avec `as`** : Force le type d'une variable, mais ne modifie pas les données internes.
- **Casting avec `<>`** : Utilise une syntaxe alternative pour forcer le type (non compatible avec TSX).
- **Force Casting** : Utilisez `unknown` pour contourner les erreurs de type et forcer un casting non valide.

## Classes en TypeScript

TypeScript améliore les classes JavaScript en ajoutant des types et des modificateurs de visibilité pour mieux structurer et contrôler l'accès aux membres d'une classe.

### Membres de la classe : Types

Les propriétés et méthodes d'une classe sont typées avec des annotations, similaires aux variables.

```
class Person {
    name!: string;
}

const person = new Person();
person.name = "Jane"; // Définition du type string pour `name`
```

## Membres de la classe : Visibilité

Les membres d'une classe peuvent avoir des modificateurs de visibilité. TypeScript offre trois types de modificateurs :

1. **public** (par défaut) : Accessible partout.
2. **private** : Accessible uniquement à l'intérieur de la classe.
3. **protected** : Accessible à l'intérieur de la classe et dans ses classes héritées.

```
class Person1 {
    private name: string;

    public constructor(name: string) {
        this.name = name;
    }

    public getName(): string {
        return this.name;
    }
}

const person1 = new Person1("Jane");
console.log(person1.getName()); // `name` est privé, donc non accessible directement.
```

## Propriétés de Paramètres

TypeScript permet de définir des membres directement dans le constructeur avec des modificateurs de visibilité.

```
class Person2 {
    public constructor(private name: string) {}

    public getName(): string {
        return this.name;
    }
}

const person2 = new Person2("Jane");
console.log(person2.getName()); // `name` est privé mais directement initialisé dans le constructeur.
```

## Readonly

Le mot-clé `readonly` empêche les membres d'une classe d'être modifiés après leur initialisation.

```
class Person3 {
    private readonly name: string;

    public constructor(name: string) {
        this.name = name;
    }

    public getName(): string {
        return this.name;
    }
}

const person3 = new Person3("Jane");
console.log(person3.getName()); // `name` est en lecture seule.
```

## Héritage : Implements

Les interfaces peuvent être utilisées pour définir un type que la classe doit suivre avec le mot-clé **implements**.

```
interface Shape {
    getArea: () => number;
}

class Rectangle1 implements Shape {
    public constructor(protected readonly width1: number, protected readonly height1: number) {}

    public getArea(): number {
        return this.width1 * this.height1;
    }
}
```

## Héritage : Extends

Les classes peuvent hériter d'autres classes en utilisant le mot-clé **extends**.

```
class Square extends Rectangle1 {
    public constructor(width: number) {
        super(width, width); // Appel du constructeur de Rectangle
    }
}
```

## Surcharge

Les classes héritées peuvent remplacer les membres de la classe parente. TypeScript permet d'utiliser le mot-clé **override** pour marquer explicitement cette substitution.

```

class Rectangle2 {
    public constructor(protected readonly width2: number, protected readonly height2: number) {}
    public toString(): string {
        return `Rectangle[width=${this.width2}, height=${this.height2}]`;
    }
}

class Square2 extends Rectangle2 {
    public override toString(): string {
        return `Square[width=${this.width2}]`; // Remplacement de `toString` de Rectangle
    }
}

```

## Classes Abstraites

Les classes abstraites servent de base pour d'autres classes et ne peuvent pas être instanciées directement. Elles peuvent avoir des membres non implémentés, marqués par le mot-clé `abstract`.

```

abstract class Polygon {
    public abstract getArea(): number;

    public toString(): string {
        return `Polygon[area=${this.getArea()}]`;
    }
}

class Rectangle3 extends Polygon {
    public constructor(protected readonly width3: number, protected readonly height3: number) {
        super();
    }

    public getArea(): number {
        return this.width3 * this.height3;
    }
}

```

## Résumé

- Types et visibilité** : Les membres de la classe peuvent être typés et avoir des niveaux d'accès (`public`, `private`, `protected`).
- Propriétés de paramètres** : Utilisation des modificateurs directement dans le `constructeur`.
- Readonly** : Empêche la modification des propriétés après l'initialisation.
- Héritage et Interfaces** : Permet de définir des types et d'hériter des classes avec `extends` et `implements`.
- Surcharge et classes abstraites** : Remplacez les méthodes héritées avec `override` et créez des classes de base abstraites avec `abstract`.

## Generics en TypeScript

Les **generics** permettent de créer des "variables de type" qui peuvent être utilisées dans des classes, fonctions et alias de types pour écrire du code réutilisable et flexible.

## 1. Generics dans les fonctions

Les generics rendent les fonctions plus générales et précises quant aux types utilisés et retournés.

```
function createPair<S, T>(v1: S, v2: T): [S, T] {
  return [v1, v2];
}

console.log(createPair<string, number>('hello', 42)); // ['hello', 42]
```

TypeScript peut aussi **inférer le type** des paramètres generics à partir des arguments de la fonction.

## 2. Generics dans les classes

Les generics permettent de créer des classes généralisées et réutilisables.

```
class NamedValue<T> {
  private _value: T | undefined;

  constructor(private name: string) {}

  public setValue(value: T) {
    this._value = value;
  }

  public getValue(): T | undefined {
    return this._value;
  }

  public toString(): string {
    return `${this.name}: ${this._value}`;
  }
}

const value = new NamedValue<number>('myNumber');
value.setValue(10);
console.log(value.toString()); // myNumber: 10
```

TypeScript peut inférer le type générique à partir des arguments du constructeur.

### 3. Generics dans les alias de types

Les generics rendent les types alias plus réutilisables.

```
type Wrapped<T> = { value: T };

const wrappedValue: Wrapped<number> = { value: 10 };
```

Les interfaces supportent également les generics avec une syntaxe similaire.

### 4. Valeur par défaut

Les generics peuvent être assignés à une **valeur par défaut** utilisée si aucun type explicite n'est spécifié.

```
class NamedValue1<T = string> {
    private _value: T | undefined;

    constructor(private name: string) {}

    public setValue(value: T) {
        this._value = value;
    }

    public getValue(): T | undefined {
        return this._value;
    }

    public toString(): string {
        return `${this.name}: ${this._value}`;
    }
}

const value1 = new NamedValue1('myNumber1'); // T est inféré comme string par défaut
value1.setValue('myValue');
console.log(value1.toString()); // myNumber: myValue
```

### 5. Contraintes avec extends

Les generics peuvent être contraints pour limiter les types acceptés.

```
function createLoggedPair<S extends string | number, T extends string | number>(v1: S, v2: T): [S, T] {
    console.log(`creating pair: v1='${v1}', v2='${v2}'`);
    return [v1, v2];
}
```

On peut combiner les contraintes avec des valeurs par défaut.

## Résumé

- **Generics** : Ajoutent de la flexibilité et permettent de généraliser des classes, fonctions ou types.
- **Inférence des types** : TypeScript infère automatiquement les types dans certains cas.
- **Valeurs par défaut** : Simplifient l'utilisation en cas d'absence de type explicite.
- **Contraintes** : Limitez les types acceptés pour rendre les generics plus spécifiques.

## Utility Types

Les utility types de TypeScript permettent de manipuler facilement les types courants pour résoudre des cas spécifiques. Voici une présentation des utility types les plus utilisés.

### 1. Partial

Convertit toutes les propriétés d'un type en **optionnelles**.

```
interface Point {  
    x: number;  
    y: number;  
}  
  
let pointPart: Partial<Point> = {};// `x` et `y` sont facultatifs  
pointPart.x = 10;
```

### 2. Required

Convertit toutes les propriétés d'un type en **requises**.

```
interface Car {  
    make: string;  
    model: string;  
    mileage?: number;  
}  
  
let myCar: Required<Car> = {  
    make: 'Ford',  
    model: 'Focus',  
    mileage: 12000 // `mileage` devient obligatoire  
};
```

### 3. Record

Crée un type d'objet avec des clés et des valeurs spécifiques.

```
const nameAgeMap1: Record<string, number> = {
  'Alice': 21,
  'Bob': 25
};
```

Equivalent à { [key : string] : number }.

#### 4. Omit

Supprime des clés spécifiques d'un type.

```
interface Person4 {
  name: string;
  age: number;
  location?: string;
}

const bob: Omit<Person4, 'age' | 'location'> = {
  name: 'Bob'
};
```

Omit supprime age et location du type.

#### 5. Pick

Conserve uniquement les clés spécifiées dans un type

```
interface Person5 {
  name: string;
  age: number;
  location?: string;
}

const bob1: Pick<Person5, 'name'> = {
  name: 'Bob'
};
```

Pick ne conserve que name, supprimant age et location.

#### 6. Exclude

Supprime des types spécifiques d'une union.

```
type Primitive = string | number | boolean;
const value2: Exclude<Primitive, string> = true; // `string` est exclu
```

## 7. ReturnType

Extrait le type de retour d'une fonction.

```
type PointGenerator = () => { x: number; y: number };
const point: ReturnType<PointGenerator> = {
  x: 10,
  y: 20
};
```

## 8. Parameters

Extrait les types des paramètres d'une fonction sous forme de tableau.

```
type PointPrinter = (p: { x: number; y: number }) => void;
const point1: Parameters<PointPrinter>[0] = {
  x: 10,
  y: 20
};
```

## 9. Readonly

Rend toutes les propriétés d'un type **readonly**.

```
interface Person5 {
  name: string;
  age: number;
}

const person5: Readonly<Person5> = {
  name: "Dylan",
  age: 35,
};

person5.name = "Israel"; // Erreur : Propriété readonly
```

## Résumé des utility types courants

Utility Type	Description
Partial<T>	Rend toutes les propriétés optionnelles.
Required<T>	Rend toutes les propriétés obligatoires.
Record<K, V>	Crée un type d'objet avec des clés K et des valeurs V.
Omit<T, K>	Supprime les clés spécifiées K d'un type T.
Pick<T, K>	Conserve uniquement les clés spécifiées K dans un type T.
Exclude<T, U>	Supprime les types de U dans l'union T.
ReturnType<F>	Extrait le type de retour d'une fonction F.
Parameters<F>	Extrait les types des paramètres d'une fonction F.
Readonly<T>	Rend toutes les propriétés d'un type immuables (readonly).

Ces utility types augmentent la productivité en facilitant la manipulation des types complexes tout en maintenant la sécurité du typage.

## keyof

Le mot-clé keyof dans TypeScript est utilisé pour extraire un type représentant les clés d'un type d'objet. Il génère une union des noms de clés disponibles dans l'objet spécifié.

### 1. keyof avec des clés explicites

Lorsqu'il est appliqué à un objet avec des clés définies, keyof crée un type union de ces clés.

```
interface Person6 {
  name: string;
  age: number;
}

// `keyof Person` crée un type union "name" | "age"
function printPersonProperty(person6: Person6, property: keyof Person6) {
  console.log(`Printing person property ${property}: ${person6[property]}`);
}

let person6 = {
  name: "Max",
  age: 27,
};

printPersonProperty(person6, "name"); // Printing person property name: "Max"
printPersonProperty(person6, "age"); // Printing person property age: "27"

// Erreur : la propriété doit être "name" ou "age"
// printPersonProperty(person6, "address");
```

### 2. keyof avec des index signatures

Quand il est utilisé sur un type avec une signature d'index, keyof retourne le type de l'index.

```
type StringMap = { [key: string]: unknown };

// `keyof StringMap` résout le type `string`
function createStringPair(property: keyof StringMap, value: string): StringMap {
  return { [property]: value };
}

const pair = createStringPair("keyName", "value");
console.log(pair); // { keyName: "value" }
```

## Résumé des utilisations de keyof

Cas d'utilisation	Résultat
keyof { a: string; b: number }	''a''
keyof { [key: string]: number }	string
keyof { [key: number]: boolean }	number
Avec une fonction ou une méthode	Valide uniquement les noms de clés spécifiques à l'objet.

## Avantages de keyof

- **Typage sécurisé** : Empêche l'utilisation de propriétés qui n'existent pas dans un type.
- **Flexibilité** : Permet de travailler avec des structures dynamiques comme des signatures d'index.
- **Interopérabilité** : Combinez avec d'autres utilitaires TypeScript comme Pick, Omit, ou les generics pour manipuler efficacement les types.

## Gestion de null et undefined

TypeScript offre un système robuste pour traiter les valeurs `null` et `undefined`. Lorsque l'option `strictNullChecks` est activée, TypeScript exige que chaque valeur soit explicitement définie ou marquée comme pouvant être `null` ou `undefined`.

### 1. Types null et undefined

`null` et `undefined` sont des types primitifs qui peuvent être combinés avec d'autres types.

```
let value3: string | undefined | null = null;
value3 = 'hello';           // Attribution d'une chaîne
value3 = undefined;        // Attribution de `undefined`
```

Avec `strictNullChecks` activé, TypeScript empêche l'utilisation de `null` ou `undefined` sans les inclure explicitement dans le type.

### 2. Optional Chaining

L'**optional chaining** permet d'accéder à des propriétés qui peuvent ne pas exister, en utilisant l'opérateur `?`

```

interface House {
  sqft: number;
  yard?: {
    sqft: number;
  };
}

function printYardSize(house: House) {
  const yardSize = house.yard?.sqft; // Vérifie si `yard` existe avant d'accéder à `sqft`
  if (yardSize === undefined) {
    console.log('No yard');
  } else {
    console.log(`Yard is ${yardSize} sqft`);
  }
}

let home: House = { sqft: 500 };
printYardSize(home); // Prints 'No yard'

```

### 3. Nullish Coalescence

L'opérateur `??` permet de définir une valeur par défaut uniquement si la valeur est `null` ou `undefined`.

```

function printMileage(mileage: number | null | undefined) {
  console.log(`Mileage: ${mileage ?? 'Not Available'}`);
}

printMileage(null);    // Prints 'Mileage: Not Available'
printMileage(0);       // Prints 'Mileage: 0'

```

Contrairement à `||`, `??` ne traite pas les autres valeurs falsy (comme `0` ou `false`) comme équivalentes à `null` ou `undefined`.

### 4. Null Assertion

Pour ignorer explicitement le fait qu'une valeur puisse être `null` ou `undefined`, vous pouvez utiliser l'opérateur `!`

```

function getValue(): string | undefined {
  return 'hello';
}

let value4 = getValue();
console.log('value length: ' + value4!.length); // Assertion que `value` n'est pas null/undefined

```

**⚠️ Attention :** L'utilisation excessive de `!` peut-être risquée et doit être limitée à des cas où vous êtes certain de la non-nullité.

### 5. Gestion des limites des tableaux

Même avec **strictNullChecks**, TypeScript suppose par défaut qu'un accès par index dans un tableau retourne une valeur définie (sauf si **undefined** est explicitement inclus dans le type).

Vous pouvez activer **noUncheckedIndexedAccess** pour indiquer que l'accès aux tableaux peut potentiellement retourner undefined.

```
let array: number[] = [1, 2, 3];
let value6 = array[0]; // Type `number | undefined` avec `noUncheckedIndexedAccess`
```

## Résumé

Fonctionnalité	Description
<b>strictNullChecks</b>	Empêche l'utilisation implicite de null ou undefined.
<b>Optional Chaining (?)</b>	Accès sécurisé aux propriétés qui peuvent ne pas exister.
<b>Nullish Coalescence (??)</b>	Définit une valeur par défaut uniquement pour null ou undefined.
<b>Null Assertion (!)</b>	Ignore explicitement la possibilité de null ou undefined.
<b>Array Bounds Handling</b>	Vérifie si un index de tableau est undefined avec <b>noUncheckedIndexedAccess</b> .

Avec ces outils, TypeScript permet de mieux sécuriser le code contre les erreurs liées à **null** et **undefined**.

## Mises à jour dans TypeScript 5.x

TypeScript 5.x introduit des fonctionnalités importantes pour améliorer la sécurité des types, la flexibilité et la qualité de vie des développeurs. Voici les principales nouveautés :

### Template Literal Types

Les **types littéraux de template** permettent de définir des types personnalisés basés sur des chaînes de caractères dynamiques à la compilation.

```
type Color = "red" | "green" | "blue";
type HexColor<T extends Color> = `#${string}`;

// Usage :
let myColor: HexColor<"blue"> = "#0000FF"; // Valide
// let invalidColor: HexColor<"yellow"> = "#FFFF00"; // Erreur : "yellow" n'est pas un type valide
```

