

# Annexe TypeScript pour un projet Spring Boot et Angular

## Introduction

Cette annexe fournit une vue d'ensemble complète des concepts TypeScript essentiels pour le développement d'une interface frontale Angular robuste, intégrée à un back-end Spring Boot. Le contenu inclut la syntaxe, les fonctionnalités avancées et les bonnes pratiques pour un développement efficace et sécurisé par typage.

## Types TypeScript

### Concepts Clés

- **Alias de Type** : Définir des structures de type réutilisables et complexes.
- **Polyvalence** : Prend en charge les types union, intersection et mappés pour des modèles dynamiques et réutilisables.

### Syntaxe Courante des Types

#### Exemple de Base d'Alias de Type

```
1 type Person = {  
2   name: string;  
3   age: number;  
4 };
```

### Types Union et Intersection

#### Types Union :

```
1 type Size = "small" | "medium" | "large";
```

#### Types Intersection :

```
1 type Location = { x: number } & { y: number };
```

### Tuples

```
1 type Point = [number, number];
```

## Propriétés en Lecture Seule (Read-only)

```
1 type Person = {  
2   readonly id: number;  
3   name: string;  
4 };
```

## Fonctionnalités Avancées

### Type basé sur des Valeurs :

```
1 const data = { id: 1, name: "Alice" };  
2 type DataType = typeof data; // Infère le type depuis 'data'
```

### Types Mappés :

```
1 type Subscriber<T> = {  
2   [P in keyof T]: (value: T[P]) => void;  
3 };
```

### Types Utilitaires :

- **Pick** : Sélectionne des propriétés spécifiques.

```
1 type NameAndAge = Pick<Person, "name" | "age">;
```

- **Partial** : Rend toutes les propriétés optionnelles.

```
1 type PartialPerson = Partial<Person>;
```

## Bonnes Pratiques

- Utilisez `readonly` pour les propriétés immuables.
- Exploitez les types utilitaires pour un code modulaire et réutilisable.
- Appliquez les génériques pour un design de type flexible.

## Syntaxe de Base d'une Interface

```
1 interface Person {  
2   name: string;  
3   age: number;  
4 }
```

## Fonctionnalités Avancées

- **Propriétés Optionnelles** : Utilisez la syntaxe `?` pour indiquer que certaines propriétés sont facultatives.

```
1 interface Person {  
2   name: string;  
3   age?: number; // Optionnel  
4 }
```

- **Propriétés en Lecture Seule** : Empêchez les modifications aux propriétés avec le modificateur `readonly`.

```
1 interface Person {  
2     readonly id: number;  
3     name: string;  
4 }
```

- **Extension des Interfaces** : Combinez plusieurs interfaces pour une meilleure extensibilité.

```
1 interface Employee extends Person {  
2     department: string;  
3 }
```

- **Signatures d'Index** : Utilisez des noms de propriétés dynamiques.

```
1 interface Dictionary {  
2     [key: string]: string;  
3 }
```

## Bonnes Pratiques

- Utilisez les interfaces pour définir des contrats d'objet.
- Exploitez `readonly` pour imposer l'immuabilité.
- Étendez les interfaces pour une meilleure extensibilité et réutilisabilité.

## Classes TypeScript

### Introduction

Les classes TypeScript étendent la syntaxe des classes JavaScript en ajoutant des types statiques, des modificateurs d'accès, et des fonctionnalités avancées de programmation orientée objet.

### Exemple de Base d'une Classe

```
1 class Person {  
2     name: string;  
3     constructor(name: string) {  
4         this.name = name;  
5     }  
6     greet(): string {  
7         return 'Hello, ${this.name}!';  
8     }  
9 }
```

## Modificateurs d'Accès

TypeScript introduit des modificateurs d'accès pour assurer l'encapsulation :

- **public** : Accessible partout.
- **private** : Accessible uniquement dans la classe.
- **protected** : Accessible dans la classe et ses sous-classes.

**Exemple :**

```
1 class Employee {
2   private id: number;
3   protected department: string;
4   public name: string;
5
6   constructor(id: number, name: string, department: string) {
7     this.id = id;
8     this.name = name;
9     this.department = department;
10  }
11
12  getDetails(): string {
13    return `${this.name} works in ${this.department}.`;
14  }
15 }
```

## Propriétés et Méthodes Statiques

```
1 class MathUtils {
2   static pi = 3.14;
3
4   static calculateArea(radius: number): number {
5     return MathUtils.pi * radius * radius;
6   }
7 }
```

## Héritage

```
1 class Manager extends Employee {
2   constructor(id: number, name: string, department: string) {
3     super(id, name, department);
4   }
5   manage(): string {
6     return `${this.name} manages the ${this.department} department.`;
7   }
8 }
```

## Accesseurs (Getters) et Mutateurs (Setters)

Encapsulez l'accès aux propriétés pour un meilleur contrôle et une validation des données.

```

1 class Rectangle {
2   private _width: number;
3   private _height: number;
4
5   constructor(width: number, height: number) {
6     this._width = width;
7     this._height = height;
8   }
9
10  get area(): number {
11    return this._width * this._height;
12  }
13
14  set dimensions({ width, height }: { width: number; height: number }) {
15    this._width = width;
16    this._height = height;
17  }
18 }

```

## Propriétés Paramétriques (Parameter Properties)

Utilisez une syntaxe abrégée pour définir et initialiser les propriétés directement dans le constructeur.

```

1 class User {
2   constructor(public name: string, private readonly id: number) {}
3 }

```

## Bonnes Pratiques

- Utilisez **private** pour l'encapsulation des données et **readonly** pour les rendre immuables.
- Profitez des *classes abstraites* pour définir des comportements communs entre les classes.
- Appliquez les membres *statiques* pour les fonctions utilitaires et les constantes partagées.

## Analyse du Flux de Contrôle TypeScript (CFA)

### Vue d'Ensemble

L'Analyse du Flux de Contrôle (Control Flow Analysis, CFA) permet à TypeScript de restreindre dynamiquement les types en fonction de la logique du code, garantissant ainsi la sécurité des types et réduisant les erreurs au moment de l'exécution.

## Caractéristiques Clés

### Réduction des Types avec des Conditions

- **typeof** : Réduire les types pour les primitives.

```

1 const input: string | number = getInput();
2 if (typeof input === "string") {
3   console.log(input.length); // input est une cha ne ici
4 }

```

- **instanceof** : Réduire les types pour les classes ou objets.

```

1 if (input instanceof Array) {
2   console.log(input.length); // input est un tableau ici
3 }

```

- **in Operator** : Vérifier l'existence d'une propriété dans un objet.

```

1 if ("error" in response) {
2   console.log(response.error);
3 }

```

## Unions Discriminées (Discriminated Unions)

Réduisez les types à l'aide d'une propriété discriminante partagée pour gérer plusieurs types au sein d'un objet.

```

1 type Response =
2   | { status: 200; data: string }
3   | { status: 404; error: string };
4
5 const res: Response = getResponse();
6 if (res.status === 200) {
7   console.log(res.data);
8 }

```

## Gardes de Type Personnalisées (Custom Type Guards)

Créez des fonctions réutilisables pour affiner les types et assurer la sécurité des données.

```

1 function isErrorResponse(obj: any): obj is { error: string } {
2   return "error" in obj;
3 }

```

## Fonctions d'Assertion (Assertion Functions)

Utilisez des fonctions d'assertion pour modifier la portée du type ou générer une erreur lorsque les conditions ne sont pas remplies.

```

1 function assertIsNumber(value: any): asserts value is number {
2   if (typeof value !== "number") {
3     throw new Error("Not a number");
4   }
5 }

```

## Bonnes Pratiques

- Utilisez les **unions discriminées** pour les objets ayant des propriétés partagées afin de gérer plusieurs types.
- Écrivez des **gardes de type personnalisées** pour une logique de réduction des types réutilisable et lisible.
- Appliquez les **fonctions d'assertion** pour une validation stricte des types lors de l'exécution.

# Résumé des Fonctionnalités et Bonnes Pratiques TypeScript

TypeScript Cheat Sheet	
<h2>Setup</h2> <p>Install TS globally on your machine</p> <pre>\$ npm i -g typescript</pre> <p>Check version</p> <pre>\$ tsc -v</pre> <p>Create the tsconfig.json file</p> <pre>\$ tsc --init</pre> <p>Set the root (to compile TS files from) and output (for the compiled JS files) directories in tsconfig.json</p> <pre>"rootDir": "./src", "outDir": "./public",</pre>	
<h2>Compiling</h2> <p>Compile a specified TS file into a JS file of the same name, into the same directory (i.e. index.ts to index.js).</p> <pre>\$ tsc index.ts</pre> <p>Write tsc to compile specified file whenever a change is saved by adding the watch flag (-w)</p> <pre>\$ tsc index.ts -w</pre> <p>Compile specified file into specified output file</p> <pre>\$ tsc index.ts --outfile output/script.js</pre> <p>If no file is specified, tsc will compile all TS files in the "rootDir" and output in the "outDir". Add -w to watch for changes.</p> <pre>\$ tsc -w</pre>	
<h2>Strict Mode</h2> <p>In tsconfig.json, it is recommended to set strict to true. One helpful feature of strict mode is No Implicit Any.</p> <pre>// Error: Parameter 'a' implicitly has an 'any' type function logName(a) {   console.log(a.name); }</pre> <p>By @DoobleDanny</p>	
<h2>Primitive Types</h2> <p>There are 7 primitive types in JS: string, number, bigint, boolean, undefined, null, symbol.</p> <p>Explicit type annotation</p> <pre>let firstname: string = 'Danny'</pre> <p>If we assign a value (as above), we don't need to state the type - TS will infer it ("implicit type annotation")</p> <pre>let firstname = 'Danny'</pre>	<h2>Arrays</h2> <p>We can define what kind of data an array can contain</p> <pre>let ids: number[] = []; ids.push(1); ids.push("2"); // Error</pre> <p>Use a union type for arrays with multiple types</p> <pre>let options: (string   number)[]; options = [10, 'UP'];</pre> <p>If a value is assigned, TS will infer the types in the array.</p> <pre>let person = ['Delia', 48]; person[0] = true; // Error - only strings or numbers allowed</pre>
<h2>Union Types</h2> <p>A variable that can be assigned more than one type</p> <pre>let age: number   string; age = 26; age = "26";</pre>	<h2>Tuples</h2> <p>A tuple is a special type of array with fixed size &amp; known data types at each index. They're stricter than regular arrays.</p> <pre>let options: (string, number); options = ['UP', 10];</pre>
<h2>Dynamic Types</h2> <p>The any type basically reverts TS back to JS.</p> <pre>let age: any = 100; age = true;</pre>	<h2>Functions</h2> <p>We can define the types of the arguments, and the return type. Below, string could be omitted because TS would infer the return type.</p> <pre>function circle(diam: number): string {   return `Circumf = \${Math.PI * diam};` }</pre> <p>The same function as an ES6 arrow</p> <pre>const circle = (diam: number): string =&gt; `Circumf = \${Math.PI * diam};`</pre> <p>If we want to declare a function, but not define it, use a function signature</p> <pre>let sayHi: (name: string) =&gt; void;  sayHi = (name: string) =&gt; console.log('Hi ' + name);  sayHi('Danny'); // Hi Danny</pre>
<h2>Literal Types</h2> <p>We can refer to specific strings &amp; numbers in type positions</p> <pre>let direction: 'UP'   'DOWN'; direction = 'UP';</pre>	<h2>Interfaces</h2> <p>Interfaces are used to describe objects. Interfaces can always be reopened &amp; extended, unlike Type Aliases. Notice that 'name' is 'readonly'</p> <pre>interface Person {   name: string;   isProgrammer: boolean; }</pre> <pre>let p1: Person = {   name: 'Delia',   isProgrammer: false, };</pre> <pre>p1.name = 'Del'; // Error - read only</pre> <p>Two ways to describe a function in an interface</p> <pre>interface Speech {   sayHi(name: string): string;   sayBye: (name: string) =&gt; string; }</pre> <pre>let speech: Speech = {   sayHi: function (name: string) {     return 'Hi ' + name;   },   sayBye: (name: string) =&gt; 'Bye ' + name, };</pre> <p>Extending an interface</p> <pre>interface Animal {   name: string; }</pre> <pre>interface Dog extends Animal {   breed: string; }</pre> <pre>let scottieDog: Dog&lt;string&gt;[] = {   breed: 'scottish terrier',   treats: ['turkey', 'haggis'], };</pre>
<h2>Objects</h2> <p>Objects in TS must have all the correct properties &amp; value types</p> <pre>let person: {   name: string;   isProgrammer: boolean; };  person = {   name: 'Danny',   isProgrammer: true, };  person.age = 26; // Error - no age prop on person object  person.isProgrammer = 'yes'; // Error - should be boolean</pre>	<h2>Enums</h2> <p>A set of related values, as a set of descriptive constants</p> <pre>enum ResourceType {   BOOK,   FILE,   FILM, }  ResourceType.BOOK; // 0 ResourceType.FILE; // 1</pre>
	<h2>The DOM &amp; Type Casting</h2> <p>TS doesn't have access to the DOM, so we need to tell TS what type of element it is via Type Casting</p> <pre>const link = document.querySelector('a');  if (element is selected by id or class, we need to tell TS what type of element it is via Type Casting const form = document.getElementById('signup-form') as HTMLFormElement;</pre>
	<h2>Narrowing</h2> <p>Occurs when a variable moves from a less precise type to a more precise type</p> <pre>let age = getUserAge(); age // string   number  if (typeof age === 'string') {   age; // string }</pre>

Figure 1: Cheat Sheet TypeScript : Fonctionnalités et Bonnes Pratiques