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

```
type Person = {
  name: string;
  age: number;
};
```

Types Union et Intersection

Types Union :

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

Types Intersection :

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

Tuples

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

Propriétés en Lecture Seule (Read-only)

```
type Person = {
  readonly id: number;
  name: string;
};
```

Fonctionnalités Avancées

Type basé sur des Valeurs :

```
const data = { id: 1, name: "Alice" };
type DataType = typeof data; // Inf re le type depuis ‘data‘
```

Types Mappés :

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

Types Utilitaires :

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

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

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

```
    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

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

Fonctionnalités Avancées

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

```
    interface Person {
      name: string;
      age?: number; // Optionnel
    }
```

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

```
interface Person {
  readonly id: number;
  name: string;
}
```

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

```
interface Employee extends Person {
  department: string;
}
```

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

```
interface Dictionary {
  [key: string]: string;
}
```

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

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  greet(): string {
    return `Hello, ${this.name}!`;
  }
}
```

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 :

```
class Employee {
  private id: number;
  protected department: string;
  public name: string;

  constructor(id: number, name: string, department: string) {
    this.id = id;
    this.name = name;
    this.department = department;
  }

  getDetails(): string {
    return `${this.name} works in ${this.department}.`;
  }
}
```

Propriétés et Méthodes Statiques

```
class MathUtils {
  static pi = 3.14;

  static calculateArea(radius: number): number {
    return MathUtils.pi * radius * radius;
  }
}
```

Héritage

```
class Manager extends Employee {
  constructor(id: number, name: string, department: string) {
    super(id, name, department);
  }
  manage(): string {
    return `${this.name} manages the ${this.department} department.`;
  }
}
```

Accesseurs (Getters) et Mutateurs (Setters)

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

```
class Rectangle {
  private _width: number;
  private _height: number;

  constructor(width: number, height: number) {
    this._width = width;
    this._height = height;
  }

  get area(): number {
    return this._width * this._height;
  }

  set dimensions({ width, height }: { width: number; height: number }) {
    this._width = width;
    this._height = height;
  }
}
```

Propriétés Paramétriques (Parameter Properties)

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

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

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.

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

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

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

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

```
if ("error" in response) {
  console.log(response.error);
}
```

Unions Discriminées (Discriminated Unions)

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

```
type Response =
  | { status: 200; data: string }
  | { status: 404; error: string };

const res: Response = getResponse();
if (res.status === 200) {
  console.log(res.data);
}
```

Gardes de Type Personnalisées (Custom Type Guards)

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

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

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.

```
function assertIsNumber(value: any): asserts value is number {
  if (typeof value !== "number") {
    throw new Error("Not a number");
  }
}
```

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

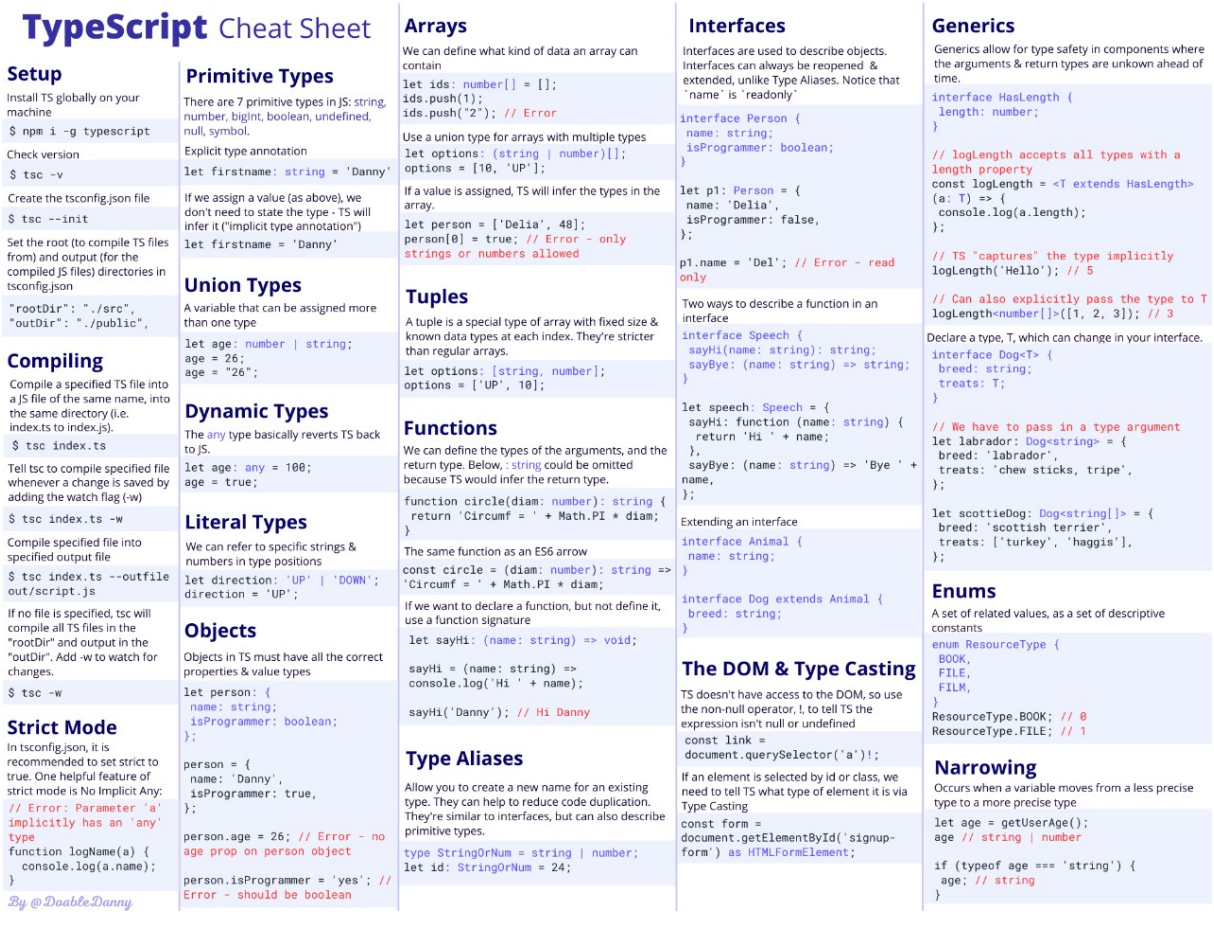

Figure 1: Cheat Sheet TypeScript : Fonctionnalités et Bonnes Pratiques