

# TypeScript Annex for Spring Boot and Angular Project

## Introduction

This annex provides a comprehensive overview of TypeScript concepts essential for developing a robust Angular front-end integrated with a Spring Boot back-end. The content includes syntax, advanced features, and best practices for type-safe and efficient development.

## TypeScript Types

### Key Concepts

- **Type Aliases:** Define reusable and complex type structures.
- **Versatility:** Supports unions, intersections, and mapped types for dynamic and reusable patterns.

### Common Type Syntax

#### Basic Type Alias Example

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

### Union and Intersection Types

#### Union Types:

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

#### Intersection Types:

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

### Tuples

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

## Read-only Properties

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

## Advanced Features

### Type from Values:

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

### Mapped Types:

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

### Utility Types:

- **Pick:** Select specific properties.

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

- **Partial:** Make all properties optional.

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

## Best Practices

- Use `readonly` for immutable properties.
- Leverage utility types for modular and reusable code.
- Apply generics for flexible type design.

## TypeScript Interfaces

### Overview

Interfaces in TypeScript define the structure of objects and serve as contracts for your data models. They enable type safety and help enforce consistency across your application.

### Basic Interface Syntax

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

## Advanced Features

- **Optional Properties:** Use the ? syntax to mark properties as optional.

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

- **Readonly Properties:** Prevent modifications to properties using the `readonly` modifier.

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

- **Extending Interfaces:** Combine multiple interfaces for scalability.

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

- **Index Signatures:** Use dynamic property names.

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

## Best Practices

- Use interfaces to define object contracts.
- Leverage `readonly` to enforce immutability.
- Extend interfaces for scalability and reusability.

## TypeScript Classes

### Introduction

TypeScript classes extend JavaScript's class syntax by adding static typing, access modifiers, and advanced object-oriented features.

### Basic Class Example

```
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 }
```

## Access Modifiers

TypeScript introduces access modifiers for encapsulation:

- **public:** Accessible from anywhere.
- **private:** Accessible only within the class.
- **protected:** Accessible within the class and its subclasses.

**Example:**

```
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 }
```

## Static Properties and Methods

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

## Inheritance

```
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 }
```

## Getters and Setters

Encapsulate property access for better control and validation.

```
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 }
```

## Parameter Properties

Shorthand for defining and initializing properties directly in the constructor.

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

## Best Practices

- Use `private` for encapsulation and `readonly` for immutability.
- Leverage *abstract classes* to define common behaviors.
- Apply *static members* for utility functions.

## TypeScript Control Flow Analysis (CFA)

### Overview

Control Flow Analysis (CFA) enables TypeScript to dynamically narrow types based on code logic, ensuring type safety and reducing runtime errors.

### Key Features

#### Type Narrowing with Conditions

- `typeof`: Narrow types for primitives.

```

1 const input: string | number = getInput();
2 if (typeof input === "string") {
3   console.log(input.length); // input is string here
4 }

```

- **instanceof:** Narrow types for classes or objects.

```

1 if (input instanceof Array) {
2   console.log(input.length); // input is an array here
3 }

```

- **in Operator:** Check for property existence in an object.

```

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

```

## Discriminated Unions

Narrow types using a shared discriminant property to handle multiple types within a single object type.

```

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 }

```

## Custom Type Guards

Create reusable functions to refine types and ensure type safety.

```

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

```

## Assertion Functions

Use assertion functions to change type scope or throw errors when a condition is unmet.

```

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

```

## Best Practices

- Use **discriminated unions** for objects with shared properties to handle multiple types.
- Write **custom type guards** for reusable and readable type narrowing logic.
- Apply **assertion functions** to enforce strict type validation during runtime.

## Summary of TypeScript Features and Best Practices

TypeScript Cheat Sheet				
<h3>Setup</h3> <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>	<h3>Primitive Types</h3> <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>	<h3>Arrays</h3> <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>	<h3>Interfaces</h3> <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> <p>The DOM &amp; Type Casting</p> <p>TS doesn't have access to the DOM, so use the non-null operator, !, to tell TS the expression isn't null or undefined</p> <pre>const link = document.querySelector('a')!;</pre> <p>If an element is selected by id or class, we need to tell TS what type of element it is via Type Casting</p> <pre>const form = document.getElementById('signup-form') as HTMLFormElement;</pre>	<h3>Generics</h3> <p>Generics allow for type safety in components where the arguments &amp; return types are unknown ahead of time.</p> <pre>interface HasLength {   length: number; }</pre> <pre>// logLength accepts all types with a length property const logLength = &lt;T extends HasLength&gt;(   a: T) =&gt; {     console.log(a.length);   }; }</pre> <pre>// TS "captures" the type implicitly logLength('Hello'); // 5</pre> <pre>// Can also explicitly pass the type to T logLength&lt;number&gt;([[1, 2, 3]]); // 3</pre> <p>Declare a type, T, which can change in your interface.</p> <pre>interface Dog&lt;T&gt; {   breed: string;   treats: T; }</pre> <pre>// We have to pass in a type argument let labrador: Dog&lt;string&gt; = {   breed: 'labrador',   treats: 'chew sticks, tripe', };</pre> <pre>let scottieDog: Dog&lt;string[]&gt; = {   breed: 'scottish terrier',   treats: ['turkey', 'haggis'], };</pre>
<h3>Compiling</h3> <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>Tell 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 out/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>	<h3>Union Types</h3> <p>A variable that can be assigned more than one type</p> <pre>let age: number   string; age = 26; age = "26";</pre> <h3>Dynamic Types</h3> <p>The any type basically reverts TS back to JS.</p> <pre>let age: any = 100; age = true;</pre> <h3>Literal Types</h3> <p>We can refer to specific strings &amp; numbers in type positions</p> <pre>let direction: 'UP'   'DOWN'; direction = 'UP';</pre>	<h3>Tuples</h3> <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> <h3>Functions</h3> <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>	<pre>interface Dog&lt;T&gt; {   breed: string;   treats: T; }</pre> <pre>// We have to pass in a type argument let labrador: Dog&lt;string&gt; = {   breed: 'labrador',   treats: 'chew sticks, tripe', };</pre> <pre>let scottieDog: Dog&lt;string[]&gt; = {   breed: 'scottish terrier',   treats: ['turkey', 'haggis'], };</pre>	<h3>Enums</h3> <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>
<h3>Strict Mode</h3> <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>	<h3>Objects</h3> <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>	<h3>Type Aliases</h3> <p>Allow you to create a new name for an existing type. They can help to reduce code duplication. They're similar to interfaces, but can also describe primitive types.</p> <pre>type StringOrNum = string   number; let id: StringOrNum = 24;</pre>	<h3>The DOM &amp; Type Casting</h3> <p>TS doesn't have access to the DOM, so use the non-null operator, !, to tell TS the expression isn't null or undefined</p> <pre>const link = document.querySelector('a')!;</pre> <p>If an element is selected by id or class, we need to tell TS what type of element it is via Type Casting</p> <pre>const form = document.getElementById('signup-form') as HTMLFormElement;</pre>	<h3>Narrowing</h3> <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>

By @DoobleDanny

Figure 1: TypeScript Cheat Sheet: Features and Best Practices