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

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

Union and Intersection Types

Union Types:

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

Intersection Types:

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

Tuples

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

Read-only Properties

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

Advanced Features

Type from Values:

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

Mapped Types:

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

Utility Types:

- **Pick:** Select specific properties.

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

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

```
    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

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

Advanced Features

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

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

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

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

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

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

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

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

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

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

```
  }
}
```

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:

```
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}.`;
  }
}
```

Static Properties and Methods

```
class MathUtils {
  static pi = 3.14;

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

Inheritance

```
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.`;
  }
}
```

Getters and Setters

Encapsulate property access for better control and validation.

```
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;
  }
}
```

Parameter Properties

Shorthand for defining and initializing properties directly in the constructor.

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

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.

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

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

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

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

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

Discriminated Unions

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

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

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

Custom Type Guards

Create reusable functions to refine types and ensure type safety.

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

Assertion Functions

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

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

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

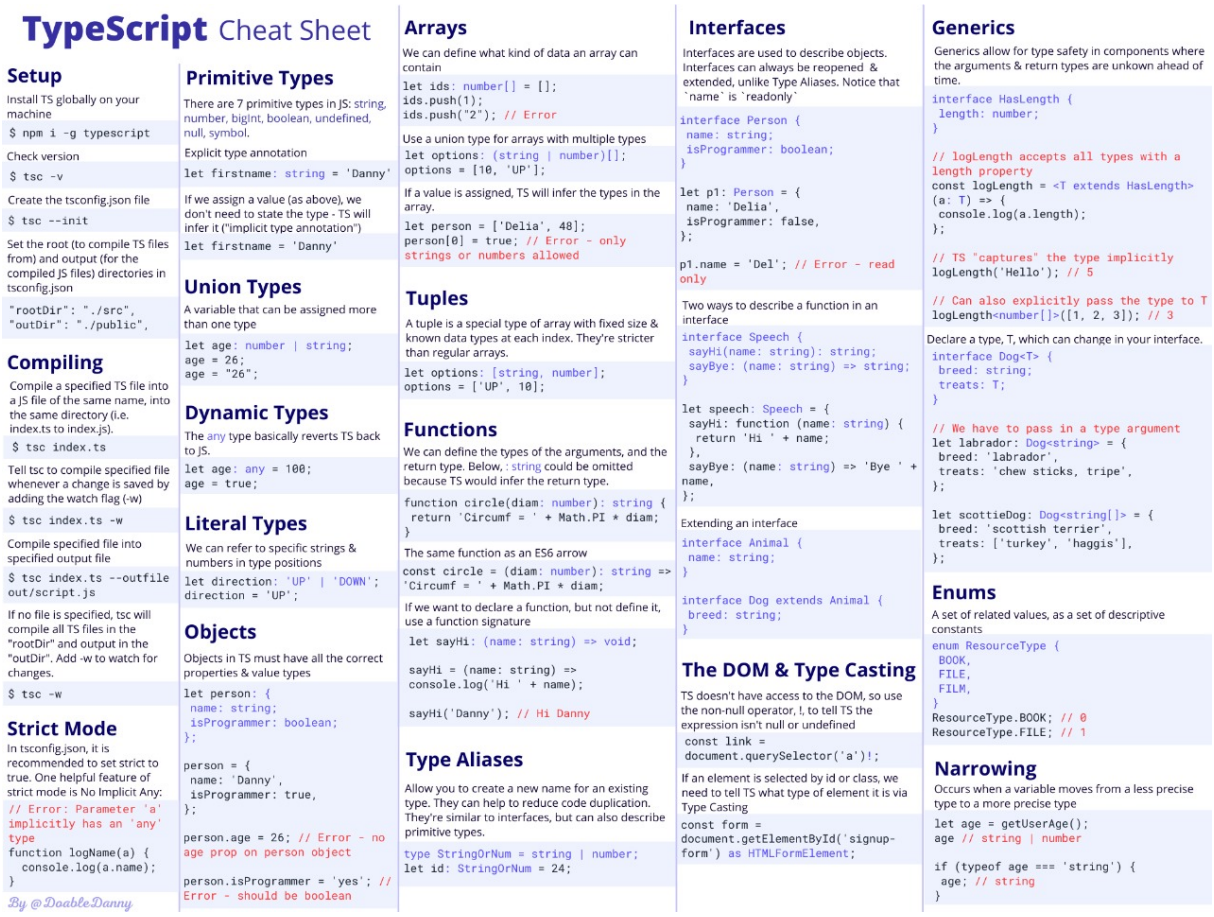

Figure 1: TypeScript Cheat Sheet: Features and Best Practices