TypeScript HANDBOOK:

A Comprehensive Guide to TypeScript

TABLE OF CONTENTS

➢ Introduction to TypeScript

o Brief Overview of TypeScript as a Statically Typed Superset of JavaScript

TypeScript is a statically typed superset of JavaScript that adds optional static types, interfaces, and other advanced features to JavaScript. Developed and maintained by Microsoft, TypeScript aims to enhance the development experience and improve the robustness of JavaScript codebases by catching errors at compile time and facilitating better tooling.

o Explanation of Why TypeScript is Used and Its Advantages Over Plain JavaScript

Why Should We Use TypeScript?

We should use TypeScript over JavaScript due to the following features:

- **Type Safety**: Helps catch type-related errors during development rather than at runtime.
- **Enhanced IDE Support**: Offers improved autocomplete, navigation, and refactoring features in modern IDEs and editors.
- **Maintainability**: Encourages cleaner, more maintainable code with explicit type declarations and interfaces.
- **Scalability**: Facilitates the development of large-scale applications by providing better structure and predictability.
- **Compatibility**: Being a superset of JavaScript, TypeScript code can seamlessly integrate with existing JavaScript libraries and projects.

➢ Getting Started

o Installation Instructions for TypeScript Compiler (tsc)

1. **Install Node.js**: Ensure Node.js is installed on your machine.
2. **Install TypeScript**: Open your terminal or command prompt and run:

   ```
   npm install -g typescript
   ```

3. **Verify Installation**: Check the installed version of TypeScript:

   ```
   tsc --version
   ```

- ## Setting Up a New TypeScript Project

1. **Create a Project Directory**

   ```
   mkdir my-typescript-project
   cd my-typescript-project
   ```

2. **Initialize a New Node.js Project**:

   ```
   npm init -y
   ```

3. **Add a TypeScript Configuration File**:

   ```
   tsc --init
   ```

   This generates a *tsconfig.json* file with default settings.

4. **Create a Source Directory**:

   ```
   mkdir src
   ```

- ## Integrating TypeScript with Existing JavaScript Projects

1. **Install TypeScript**:

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

2. **Create or Update *tsconfig.json*:**

   ```
   {
     "compilerOptions": {
       "outDir": "./dist",
       "allowJs": true,
       "checkJs": true
     },
     "include": ["src/**/*"]
   }
   ```

3. **Rename JavaScript Files**: Optionally, rename *.js* files to *.ts* or *.tsx*.
4. **Run the TypeScript Compiler**:

   ```
   Tsc
   ```

➢ Basic Syntax and Types

○ Overview of TypeScript Syntax Compared to JavaScript

TypeScript syntax extends JavaScript by adding type annotations, interfaces, and other type-related features. Here's a simple comparison:

JavaScript:

```
function greet(name) {
  return "Hello, " + name;
}
```

TypeScript:

```
function greet(name: string): string {
  return "Hello, " + name;
}
```

○ Introduction to Basic Data Types

- **Number**:

  ```
  let count: number = 5;
  ```

- **String**:

  ```
  let name: string = "Alice";
  ```

- **Boolean**:

  ```
  let isDone: boolean = false;
  ```

- **Null and Undefined**:

  ```
  let n: null = null;
  let u: undefined = undefined;
  ```

○ Understanding Type Annotations and Type Inference

- **Type Annotations**: Explicitly declare the type of a variable.

  ```
  let age: number = 30;
  ```

- **Type Inference**: TypeScript automatically infers the type based on the assigned value.

  ```
  let isActive = true; // inferred as Boolean
  ```

➢ Static Typing

○ Explanation of Static Typing and Its Benefits

Static typing allows developers to define the types of variables, function parameters, and return values. Benefits include:

- **Early Error Detection**: Catch type errors during development.
- **Improved Documentation**: Types serve as implicit documentation.
- **Enhanced Tooling**: Better autocomplete and code navigation.

○ Declaring Variable Types Using Type Annotations

Explicitly specify types for variables and function parameters.

```
let message: string = "Hello, TypeScript!";
function add(x: number, y: number): number {
  return x + y;
}
```

○ Type Inference: How TypeScript Infers Types Based on Context

TypeScript infers types from initial values or context, reducing the need for explicit annotations.

```
let greeting = "Hello, World!"; // inferred as string
let total = add(5, 10); // inferred as number
```

➢ Interfaces

○ Definition and Usage of Interfaces in TypeScript

Interfaces define the shape of objects, ensuring they adhere to a specific structure.

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

○ Creating Interfaces for Object Shapes and Contracts

Interfaces can be used to type-check objects.

```
const person: Person = {
  name: "John",
  age: 25,
};
```

○ Optional Properties and Read-Only Properties in Interfaces

- **Optional Properties**:

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

- **Read-Only Properties**:

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

➢ Classes

○ Object-Oriented Programming Concepts in TypeScript

Object-Oriented Programming (OOP) in TypeScript brings the principles of classical OOP—such as encapsulation, inheritance, and polymorphism—into the JavaScript world.

TypeScript supports object-oriented programming with classes, inheritance, and encapsulation.

TypeScript's type system and syntax make it a robust choice for developers looking to build large-scale, maintainable applications.

○ Defining Classes with Properties and Methods

```
class Animal {
  name: string;

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

  makeSound(): void {
    console.log(this.name + " makes a sound.");
  }
}
```

○ Constructors and Access Modifiers (public, private, protected)

- **Constructor**:

  ```
  class Animal {
    constructor(public name: string) {}
  }
  ```

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

```
class Animal {
  private age: number;
```

```
  constructor(public name: string, age: number) {
    this.age = age;
  }

  protected getAge(): number {
    return this.age;
  }
}
```

- **Inheritance and Method Overriding**

```
class Dog extends Animal {
  constructor(name: string) {
    super(name);
  }

  makeSound(): void {
    console.log(this.name + " barks.");
  }
}
```

➢ Generics

- **Introduction to Generics in TypeScript**

Generics allow for creating reusable components that work with any data type.

```
function identity<T>(arg: T): T {
  return arg;
}
```

- **Creating Reusable Components with Generic Types**

```
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}
```

```
let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) {
  return x + y;
};
```

- **Using Generic Constraints to Enforce Type Relationships**

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
```

➢ Advanced TypeScript Concepts

- **Union Types and Intersection Types**
  - **Union Types**:

    ```
    let value: string | number;
    value = "hello";
    value = 42;
    ```

  - **Intersection Types**:

    ```
    interface A {
      x: number;
    }

    interface B {
      y: number;
    }
    ```

```
let obj: A & B = { x: 1, y: 2 };
```

Type Aliases and Type Assertions

- **Type Aliases**:

```
type StringOrNumber = string | number;
```

- **Type Assertions**:

```
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```

Type Guards for Working with Unions

Type guards help narrow down types within union types.

```
function isNumber(x: any): x is number
 {
  return typeof x === "number";
}

function example(x: string | number)
 {
  if (isNumber(x))
{
    // x is a number here
    console.log(x.toFixed(2));
  }
else
 {
    // x is a string here
    console.log(x.toUpperCase());
  }
}
```

Understanding Conditional Types and Mapped Types

- **Conditional Types**:

```
type NonNullable<T> = T extends null | undefined ? never : T;
```

- **Mapped Types**:

```
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

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

type ReadonlyPerson = Readonly<Person>;
```