***TypeScript Handbook***

## **Introduction to TypeScript**

### **Brief Overview of TypeScript as a Statically Typed Superset of JavaScript**

TypeScript is an open-source programming language developed and maintained by Microsoft. It is a statically typed superset of JavaScript that adds optional static typing, classes, and interfaces. TypeScript code is transpiled to plain JavaScript, ensuring compatibility with any JavaScript engine.

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

1. **Static Typing:** Helps catch errors at compile time, improving code quality and maintainability.
2. **Enhanced IDE Support:** Provides better autocompletion, navigation, and refactoring capabilities.
3. **Modern JavaScript Features:** Includes features from the latest ECMAScript standards and additional type-checking capabilities.
4. **Improved Readability and Maintainability:** Type annotations make it easier to understand what types of values are expected.
5. **Interoperability:** Works seamlessly with existing JavaScript codebases, making it easy to gradually adopt TypeScript.

## **Getting Started**

### **Installation Instructions for TypeScript Compiler (tsc)**

To install TypeScript, you need Node.js and npm installed on your machine. Once you have them, you can install the TypeScript compiler globally using npm:

npm install -g typescript

Verify the installation by checking the TypeScript version:

tsc --version

### **Setting Up a New TypeScript Project**

1. **Initialize a new Node.js project**:  
     
   mkdir my-project

cd my-project

npm init -y

1. **Install TypeScript and create a tsconfig.json file**:  
     
   npm install typescript --save-dev

tsc --init

1. **Create a sample TypeScript file**:  
     
   // src/index.ts

const greeting: string = "Hello, TypeScript!";

console.log(greeting);

1. **Compile the TypeScript file**:

tsc

### **Integrating TypeScript with Existing JavaScript Projects**

To integrate TypeScript into an existing JavaScript project:

1. **Add TypeScript as a dev dependency**:

npm install typescript --save-dev

1. **Rename JavaScript files to TypeScript files (.ts)**.
2. **Add type annotations and resolve type errors**.
3. **Create or update tsconfig.json to include necessary compiler options**.

## **Basic Syntax and Types**

### **Overview of TypeScript Syntax Compared to JavaScript**

TypeScript syntax is very similar to JavaScript, with additional type annotations and interfaces.

**JavaScript**:

function greet(name) {

return "Hello, " + name;

}

**TypeScript**:

function greet(name: string): string {

return `Hello, ${name}`;

}

### **Introduction to Basic Data Types**

TypeScript supports the following basic data types:

* number: Represents both integer and floating-point numbers.

let num: number = 42;

* string: Represents text data.

let str: string = "Hello, TypeScript";

* boolean: Represents true or false values.

let isDone: boolean = true;

* null and undefined: Represent the absence of a value.

let n: null = null;

let u: undefined = undefined;

### **Understanding Type Annotations and Type Inference**

* **Type Annotations:** Explicitly specify the type of a variable.  
   let count: number = 10;
* **Type Inference:** TypeScript infers the type based on the value assigned.  
   let inferred = "This is a string"; //TypeScript infers `string` type.

## **Static Typing**

### **Explanation of Static Typing and Its Benefits**

Static typing involves defining variable types at compile time, which helps catch errors early, improves code quality, and enhances development tools' effectiveness.

### 

### **Declaring Variable Types Using Type Annotations**

let isDone: boolean = false;

let height: number = 6;

let name: string = "John";

### **Type Inference: How TypeScript Infers Types Based on Context**

TypeScript can infer types when initial values are assigned, providing type safety without explicit annotations.

let score = 100; // inferred as number

## **Interfaces**

### **Definition and Usage of Interfaces in TypeScript**

Interfaces define the structure of objects, ensuring they meet certain requirements.

interface Person {

name: string;

age: number;

}

const john: Person = {

name: "John Doe",

age: 30

};

### **Creating Interfaces for Object Shapes and Contracts**

Interfaces can be used to define the expected shape of objects, making the code more predictable.

interface User {

name: string;

age: number;

email?: string; // Optional property

}

### **Optional Properties and Read-Only Properties in Interfaces**

* **Optional Properties**: Use ? to denote optional properties.  
  interface Car {

make: string;

model: string;

year?: number; // optional

}

* **Read-Only Properties**: Use readonly to make properties immutable.  
  interface Book {

readonly title: string;

author: string;

}

## **Classes**

### **Object-Oriented Programming Concepts in TypeScript**

TypeScript supports object-oriented programming principles, such as encapsulation, inheritance, and polymorphism.

### **Defining Classes with Properties and Methods**

class Animal {

name: string;

constructor(name: string) {

this.name = name;

}

move(distance: number): void {

console.log(`${this.name} moved ${distance} meters.`);

}

}

### **Constructors and Access Modifiers (public, private, protected)**

* **Public**: Accessible from anywhere.
* **Private**: Accessible only within the class.
* **Protected**: Accessible within the class and its subclasses.

class Person {

private name: string;

constructor(name: string) {

this.name = name;

}

public getName(): string {

return this.name;

}

}

### **Inheritance and Method Overriding**

Classes can inherit from other classes, and methods can be overridden to provide specific implementations.

class Dog extends Animal {

bark(): void {

console.log("Woof! Woof!");

}

move(distance: number = 10): void {

console.log("Running...");

super.move(distance);

}

}

## **Generics**

### **Introduction to Generics in TypeScript**

Generics provide a way to create reusable components with types that can be specified later.

function identity<T>(arg: T): T {

return arg;

}

let output = identity<string>("Hello, Generics!");

### **Creating Reusable Components with Generic Types**

Generics make components more flexible and reusable.

class GenericNumber<T> {

zeroValue: T;

add: (x: T, y: T) => T;

}

let myGenericNumber = new GenericNumber<number>();

myGenericNumber.zeroValue = 0;

myGenericNumber.add = (x, y) => x + y;

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

Generic constraints ensure that a type meets certain criteria.

interface Lengthwise {

length: number;

}

function logLength<T extends Lengthwise>(arg: T): T {

console.log(arg.length);

return arg;

}

logLength({ length: 10, value: "Hello" });

## **Advanced TypeScript Concepts**

### **Union Types and Intersection Types**

* **Union Types**: A value that can be one of several types.  
  typescript  
  Copy code  
  let value: number | string;

value = 42;

value = "Hello";

* **Intersection Types**: Combines multiple types into one.  
  typescript  
  Copy code  
  interface ErrorHandling {

success: boolean;

error?: { message: string };

}

interface ArtworksData {

artworks: { title: string }[];

}

type ArtworksResponse = ArtworksData & ErrorHandling;

### **Type Aliases and Type Assertions**

* **Type Aliases**: Create a new name for a type.  
   type StringOrNumber = string | number;
* **Type Assertions**: Override TypeScript's inferred type.  
   let someValue: any = "This is a string";

let strLength: number = (someValue as string).length;

### **Type Guards for Working with Unions**

Type guards narrow down the type within a conditional block.

function isString(value: unknown): value is string { return typeof value === "string";

}

let value: unknown = "Hello";

if (isString(value)) { console.log(value.toUpperCase()); // value is now a string

}

### **Understanding Conditional Types and Mapped Types**

* **Conditional Types**: Types that depend on a condition.

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

* **Mapped Types**: Create new types by transforming existing ones.

type Readonly<T> = {

readonly [P in keyof T]: T[P];

};

type Partial<T> = {

[P in keyof T]?: T[P];

};