TypeScript

1. **Overview**

* **What and why?**
* **Environment Setup**
* **Variable declarations**
* **Variable Types**
* **Functions**
* **Interface**
* **Class**
* **Access Modifiers**

1. **TypeScript**
2. **What and Why**

**What**

**Open-source programming language from Microsoft**

**Typed superset of Javascript**

**Compiles down to plain Javascript**

**Why**

**Relation to JS**

**Optional static typing and type inference** 🡺(JS is dynamically typed which means JS does not Know what type a variable is until it is actually instantiated at runtime which is to late Typescript on the other hand add type support to JS so with Typescript we can identified errors when we type the code )

**IDE support (**we get intelligent support ability to refactor the code easily and red squiggly line whenever there is an error along with the description of that ever**)**

**Rapid growth and use (** Is the main programming language of angular it has found its way into React and View as well ,Microsoft is working on the Project and it will improve it**)**

1. **Environment Setup**
2. First we need to install nodeJs
3. Install Typescript (npm I -g typescript)
4. npx tsc -v (To see the version)

npx tsc main.ts (create the file main.js which contain the transpiled code)

npx tsc main --watch(automatically recompile that typescript file whenever is a change)

So we have a typescript file wich get transpired to a js file which is then use in our application

1. **Variable declarations**

By making use of left and const Keywords for variable declarations typeScript helping us to reduce the bugs in our codeIn

export {}

let message='Welcome Back'

console.log(message)

let x=10;

const y=5

// let x=5; we can't

// const z we can't

In typeScript we can use template string and template string we can use multiple lines and have embeddes expressions

let sentence= `Muy name is ${name}

i am a beginner in Typescript`;

console.log(sentence);

1. **Variable Types**

Boolean Number String 🡺 Is the basic types

Null Undefined 🡺 Null and undefined are types and subTypes

* 1. Definition

In TypeScript, a "type" refers to the specific kind or category of value that a variable or expression can hold. TypeScript includes built-in types such as number, string, and boolean, as well as more complex types like arrays and objects. Additionally, you can create your own custom types using interfaces or type aliases. Declaring the type of a variable or expression helps to catch errors early in the development process and can make your code more readable and maintainable.

export {};

let isBeginner: boolean = true;

let tota: number = 0;

let name: string = "Vishwas";

* 1. Important things using types

The first one is type checking Example:

//name =true  don't work because name is a string

The second one is intelligence

//name.  so when we write name. we can see all the properties

* 1. Null and undefined

let n: null = null;

let u: undefined;

undefined and null are classiffed as subtypes of all other types that mean

let isNewx: number = null;

let muName: string = undefined;

* 1. Array

let List1: number[] = [1, 2, 3];

let list2: Array<number> = [1, 2, 3];

Mix type array it's possible but: (the number of element are fixed and the order of values has to match the order of types)

let person1: [string, number] = ["Chris", 22];

* 1. Enum Type

Is a way of giving more friendly names to a set of numeric values decalring an Enum

enum Color0 {

  Red,

  Green,

  Blue,

}

enum Color1 {

  Red = 5,

  Green,

  Blue,

}

let c0: Color0 = Color0.Green;

let c1: Color1 = Color1.Green;

console.log(c0, c1); //1,6

* 1. Any Type

We use it when:

* we know that the value is dynamic(the type change (user input))
* helpful wen we are just migrating from JS to TS
* helps to assign different types of values

let randomValue: any = 10;

randomValue = true;

randomValue = "Houssam";

//AnyTyoe don't verify anithing

console.log(randomValue.name);

randomValue();

randomValue.toUpperCase();

* 1. Unknow Type

Is very simmillar to the any type but we can't acces any propereties of an unknown type nor can we call or contruect them

let randomValue1: unknown = 10;

randomValue1 = true;

randomValue1 = "Houssam";

function hasName(obj: any): obj is { name: string } {

  return !!obj && typeof obj === "object" && "name" in obj;

}

if (hasName(randomValue1)) {

  console.log(randomValue1.name);

}

// we nedd to use a type assertion to convert the type sysytem that we know

(randomValue1 as string).toUpperCase();

* 1. Type inference

let a

a=10

a=true

let b=20

// b=true error because b is a number that is a type inference

* 1. Union of types

**Ability to specify a union of types for the same variable**

let multiType : number | boolean

multiType=20

multiType=true

1. **Functions**
   1. Types Function

**In TypeScript we can specify types for function parameters**

function add(num1:number,num2:number):number{

    return num1+num2

}

add(5,10)

// add(5,'10') dn't work

* 1. **Optional parameter and default parameter**

**we just add ? to the end of the parameter**

function add1(num1:number,num2?:number):number{

    if(!num2){

        return num1

    }

    else{

    return num1+num2

    }

}

add1(5)

//default parametre

function add2(num1:number,num2:number=10):number{

   return num1+num2

}

add2(5,15)

add2(5)

1. **Interface**

// In TypeScript, an interface defines a contract for the shape of an object. It specifies the properties and methods that an object must have, but does not provide an implementation for them. Interfaces are used to define a structure for objects, classes, and other types, and can be used for type checking at compile-time.

// Here is an example of an interface definition in TypeScript:

// Copy code

interface Point {

    x: number;

    y: number;

}

// In this example, the Point interface defines two properties, x and y, both of which are numbers. Any object that implements this interface must have these two properties with the specified types.

// You can also use interfaces to define the shape of function types. Here is an example:

// Copy code

interface SearchFunc {

    (source: string, subString: string): boolean;

}

// In this example, the SearchFunc interface defines a single method, which takes two strings as arguments and returns a boolean. Any function that implements this interface must have the same signature.

// You can use interfaces to create a class.

// Copy code

interface Point {

    x: number;

    y: number;

}

class MyPoint implements Point {

    x: number;

    y: number;

    constructor(x: number, y: number) {

        this.x = x;

        this.y = y;

    }

}

// In this example, MyPoint is a class that implements the Point interface. It must have two properties x and y of type number.

interface Person{

      firstName:string

      lastName:string

      age?:number//optional

}

function fullName(person: Person){

    console.log(`${person.firstName} ${person.lastName}`)

}

let p={

    firstName:'Bruce',

    lastName:'wayne',

    test:'test'

}

fullName(p)

1. **Classes and access Modifiers**
   1. Class

**helps us to build application using the object-oriented class-based the syntax is familliar to java**

class Employee {

    employeeName:string;

    constructor(name:string){

        this.employeeName=name

    }

    greet(){

        console.log(`Good Morning ${this.employeeName}`)

    }

}

let emp1=new Employee('Vishwaas')

emp1.greet()

* 1. Inhertince classes

1. class Manager extends Employee{
2. constructor(managerName:string){
3. super(managerName)
4. }
5. delegateWork(){
6. console.log(`Manager delegating tasks`)
7. }
8. }
9. let m1=new Manager('Bruc')
10. m1.greet()
11. m1.delegateWork()
    1. Inhertince classes

* Acces modifiers are basically keywords that set the accessibility of properties and methods in a class
* JavaScript does not have built-in acces modifiers (such as private, protected, and public)
* By default each class member is public
* Private it cannot be acces from outside of its containning class
* Private it cannot be acces even in the derived class
* Protected accessibilittt within the class the classes derived fron it

class Employee1 {

    private employeeName:string;

    constructor(name:string){

        this.employeeName=name

    }

    greet(){

        console.log(`Good Morning ${this.employeeName}`)

    }

}

class Manager1 extends Employee1{

    constructor(managerName:string){

        super(managerName)

    }

    delegateWork(){

        // console.log(`Manager delegating tasks ${this.employeeName}`)

        //impossible because employeeName is private

    }

}

let emp2=new Employee1('Vishwaas')

// console.log(emp2.employeeName) don't work because employeeName is private

//We can use the protected modifiers to acces to the derived class

class Employee2 {

    protected employeeName:string;

    constructor(name:string){

        this.employeeName=name

    }

    greet(){

        console.log(`Good Morning ${this.employeeName}`)

    }

}

class Manager2 extends Employee2{

    constructor(managerName:string){

        super(managerName)

    }

    delegateWork(){

        console.log(`Manager delegating tasks ${this.employeeName}`)

    }

}

let emp3=new Employee2('Vishwaas')

// console.log(emp3.employeeName) don't work because employeeName is protected

//Public for free accessibilitty

// Private accessibilittt within the class

// Protected accessibilittt within the class the classes derived fron it