**Typescript**

What is typescript?

Typescript is a superset of JavaScript that adds static typing and other features to enhances JavaScript development. It was developed by Microsoft on oct 1, 2012, and is now an open-source project with a large community of contributors.

Any valid JavaScript code is also valid typescript code.

* ADVANTAGES: Easier to Read , Write and Maintain

EG:

|  |  |
| --- | --- |
| In Javascript:  let fname=”Vinod”  Fname =105  Fname = true;  O/P : true | In Typescript:  let fname : string =”Vinod”  Fname =105  Fname = true;  O/P : Error |

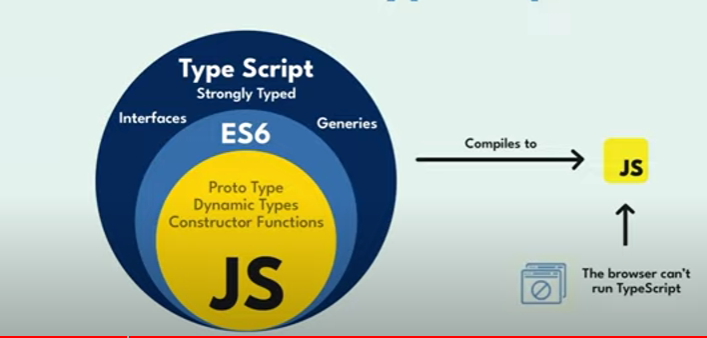
let fname : string = “vinod” // static typing or Type annotations

fname = 105 // Error due Type Inference

Interfaces, Tuples, Enums, Generic, Classes. Etc. – Extra features in Typescript

TypeScript catches errors at compile time instead of runtime, which makes debugging easier and reduces the likelihood of bugs in production.

DISADVANTAGE: Browser can’t support typescript



**Tutorial 2:** Installation

Intall Node.js LTS and Check in cmd whether node.js is installed or not using command node ~v

Install Visual studio code, In terminal type npm install -g typescript

tsc - -v to check typescript installed or not

Ep 3: TS Configuration file:

To avoid creating js file when ts file have error code we need tsconfig.file

To create tsconfig.file: tsc –init

"noEmitOnError": true,   🡪 tsc

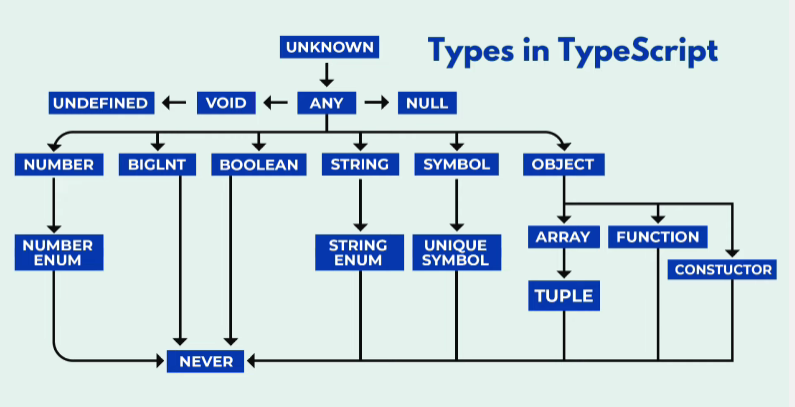
Disable emitting files if any type checking errors are reported.

tsc --noEmitonError index.ts

Tutorial 4: Type Annotations in TypeScript

Type Annotations is a way of explicitly specifying the type of a variable, function parameter, or function return value.

Eg: let myfavnum : number =10;



Advantages:

* It helps the typescript compiler to enforce type checking and provide type safety during development.
* What operations could we performed on that variable or value

Eg:

1: Substring:

let longText: string = "This is a long sentence.";

let shortText: string = longText.substring(0, 10);

2: String Comparison:

let str1: string = "Hello";

let str2: string = "World";

let areEqual: boolean = str1 === str2;

3: String Template:

let product: string = "Phone";

let price: number = 500;

let message: string = `The product ${product} is priced at ${price} dollars.`;

Any javascript code is valid typescript code

console.log("DATA");

let a= 5;

console.log(a);

BigInt: It is introduced in 2020 typescript version.

* It is build- in type that allow you to work with numbers that are larger than the range supported by the regular number type i.e 2^53
* BigInt literals are written by appending the n suffix to an integer literal.

let big:number=Number.MAX\_SAFE\_INTEGER;

console.log(big);

Eg: let bignumber:BigInt=66666666666666666666666n;

let big:number=(Number as any).MAX\_SAFE\_INTEGER;

console.log(big);

Any Type: The any type is the most flexible type in ts. It essentially turns off all type checking for the variables or expressions it is applied to.

Let myfavnum: any =5;

myfavnum=”vinod”;

Usecases:

* Working with Dynamic data
* Migration from Javascript

Unknown Type: It is a safer alternative to any because it still enforces type checking and type safety.

let myfavnum=55;

myfavnum=true //error type checking

Variables of type unknown can hold values of any type, but you must perform type checks or type assertions before using them in specific ways.

let num2: unknown;

num2 = 5;

num2 = "thapa";

//num2 =true;

if(typeof num2 === "number"){

    console.log("Number ")

    console.log(num2);

}

console.log(num2);

PS E:\Typescript> tsc --noEmitonError index.ts

PS E:\Typescript> node index.js

thapa

PS E:\Typescript>

Eg:



Type inference : It ability of the typescript compiler to automatically determine and assign types to variables, expressions, and function return values based on their usage and context in the code.

let myName =”vivek”;

Function :

function greet(name:string,id:number) // parameters{

    console.log(`welcome, ${name} and your id is ${id}`);

}

greet("vivek",1); //arguments

Fat Arrow Function

const greet=(name:string,id:number)=>{

    console.log(`welcome, ${name} and your id is ${id}`);

}

greet("vivek",1);

const greet=(name:string,id:number):string =>{

    return `welcome, ${name} and your id is ${id}`;

}

const mygreet=greet("vivek",1);

console.log(mygreet);

optional parameter: It denoted by appending a ? symbol after the parameter name

const greet=(name:string ,id?:number):string => {

    if(id){

        return `welcome, ${name}`;

    }

    return `welcome, ${name} and your id is ${id}`;

}

const mygreet=greet("vivek",5);

console.log(mygreet);

const mygreet1=greet("vivek");

console.log(mygreet1);

PS E:\Typescript> tsc --noEmitonError index.ts

PS E:\Typescript> node index.js

welcome, vivek

welcome, vivek and your id is undefined

PS E:\Typescript>

Default parameter: Specified by providing a default value in the parameter declaration.

Array:

To initialize arrays using varous approaches.

1. Using square brackets:

Const numbers : number[] = [1,2,3,4];

1. Using the array contructor:

Const numbers1: number[] = new Array{1,2,3};

1. Using the Array of method:

Const name: string[] = Arrey.of(“vinod”,”vivek”);

Methods:

const fruits:string[]= ["apple","banana"];

fruits.push("mango");

console.log(fruits);

fruits.pop();

console.log(fruits);

PS E:\Typescript> tsc --noEmitonError index.ts

PS E:\Typescript> node index.js

[ 'apple', 'banana', 'mango' ]

[ 'apple', 'banana' ]

Iterations:

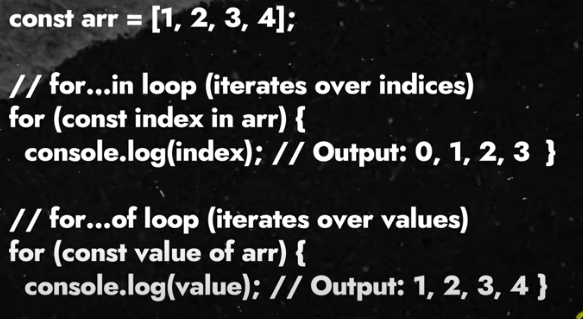
For loop:

for(let i=0;i<fruits.length;i++) {

       console.log(fruits[i]);

}

For in and For of:



Foreach() :

fruits.forEach((curVal:String) => console.log(curVal));

Map Methods: It creates a new array by applying a provided function to each element of the original array. It transforms each element and returns a new array with the transformed values.

const numbers : number[] = [1,2,3,4];

const doubleData :number[] = numbers.map((curVal:number) => curVal\*2);

console.log(doubleData);

Filter Method:

var Evendata:number[] = numbers.filter((curVal:number)=> curVal%2 ==0 );

console.log(Evendata);

Objects: In Typescript, objects are used to represent data with key-value pairs. Each key in the object is a string that maps to a value.

const person:{

    name: String;

    age: number;

    address: {city:string ;country:string}

} = {

    name:'vivek',

    age:25,

    address:{

        city:'Pune',

        country: 'India'

    }

};

console.log(person.address.city);

Type Aliases: Name of alias var should start with capital letter.

type Person={

    name: String;

    age: number;

    class? :string; //optional parameter

    address: {city:string ;country:string}

}

const person:Person= {

    name:'vivek',

    age:25,

    class:'V',

    address:{

        city:'Pune',

        country: 'India'

    }

};

const person2:Person= {

    name:'vinod',

    age:25,

    address:{

        city:'Chennai',

        country: 'UK'

    }

};

console.log(person.address.city);

console.log(person2.address.city);

Call Signatures: The function call signature refers to the declaration or definition of a function, which includes the function’s name, parameters, and return type. It defines the structure and type information of a function without including the function’s implementation or body.

There are typically used inside object type notation to describe the shape of function’s within object types.

type Student={

    name:string;

    age:number;

    gender?: string;

    greet:(country:string) =>string // call signature

(country:string) =>string // Pure call signature

}

const student1:Student ={

    name:'vivek',

    age: 29 ,

    greet: (country): string => `welcome my name is ${student1.name}, I am ${student1.age} yrs old & I  am from ${country}`

}

const intro:(student1: Student)=>string =(student1:Student):string =>{

    const {name,age}=student1;

    return `welcome my name is ${name}, I am ${age}yrs old`;

}

console.log(intro(student1));

console.log(student1.greet('India'));

console.log(student1(country:'India'));

Enums:

enum Roles{

    user="user",admin="admin"

}

type loginDetails={

    name?:string;

    email:string;

    password:string;

    role:Roles

}

const user1: loginDetails={

    email:"vinod@gmail.com",

    password:"password",

    role:Roles.admin

}

const user2: loginDetails={

    email:"vivek@gmail.com",

    password:"password",

    role:Roles.user

}

const isAdmin:(user1:loginDetails)=> void=(user1:loginDetails):string=>{

    const{name,email,role}=user1;

    return role === "admin" ? `${email} is allow to edit the website`:`${email} is not allow to edit the website`;

}

console.log(isAdmin(user1));

console.log(isAdmin(user2));

PS E:\Typescript> tsc --noEmitonError index.ts

PS E:\Typescript> node index.js

vinod@gmail.com is allow to edit the website

vivek@gmail.com is not allow to edit the website

Tuples: Tuples are a data structure that allows you to store a fixed-size collection of elements of different types.

They are similar to arrays, But with a key difference : the types of elements in a tuple are fixed and declared at the time of creation.

type Personinfo= readonly [string,number,boolean]

const displayPersonInfo:(person:Personinfo)=>void=

(person:Personinfo):void=>{

     const[name,age,hasDriverLicense]=person;

     console.log(`Name:${name},Age: ${age},Driver's License: ${hasDriverLicense ? "Yes" : "No"}`);

}

const person1:Personinfo=['vinod',29,true];

const person2:Personinfo=['Vivek',30,false];

displayPersonInfo(person1);

displayPersonInfo(person2);

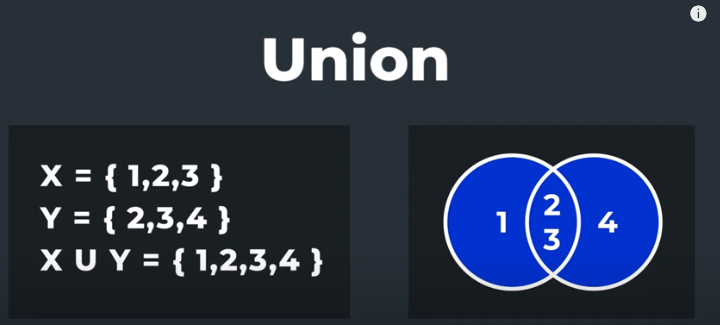
PS E:\Typescript> node index.js

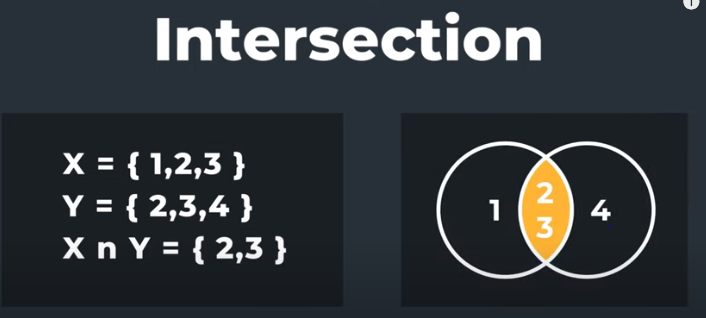
Name:vinod,Age: 29,Driver's License: Yes

Name:Vivek,Age: 30,Driver's License: No

Readonly Property: Block adding new elment in person1 and person2.

Unions and Intersections:



I

Union Types allow you to specify that a variable can hold values of multiple types. You use the | (pipe) symbol to define a union type.

const inputValue=(value:string | number|boolean)=>{

   if(typeof value === 'number'){

    return value\*2;

   }else if(typeof value === "string"){

    return value.toUpperCase();

   }else{

    throw new Error('Invalid input data');

   }

}

console.log(inputValue(10));

console.log(inputValue(true));

console.log(inputValue('HelloWorld'));

Intersection types allow you to combine multiple tupe into a single type. You use the & (ampersand) symbol to define an intersection type.

type person={

   name:string;

   age:number

}

type Employee={

   emp\_id:number;

   department:string;

}

type EmployeeDetatils = person  & Employee;

const employee:EmployeeDetatils={

   name:'vinod',

   age:29,

   emp\_id:66,

   department:'IT'

}

Generics: It allow you to create reusable components or functions that can work with multiple data types.

function logandreturn<T>(value:T):T{

   return value;

}

const num=logandreturn(42);

const str=logandreturn("Hello");

const bool=logandreturn(true);

console.log(num);

console.log(str);

console.log(bool);

PS E:\Typescript> node index.js

42

Hello

true

best practice: Declare type while calling the function

const num=logandreturn<number>(42);

const str=logandreturn<string>("Hello");

const bool=logandreturn<boolean>(true);

Multiple type variable in Generics:

function logandreturn<T,U>(a:T,b:U){

    console.log(typeof a);

    console.log(typeof b);

}

const num= logandreturn<number,string>(42,"Ram");

const str=logandreturn<string,number>("Hello",55);

const bool=logandreturn<boolean,string>(true,"hello");

PS E:\Typescript> node index.js

number

string

string

number

boolean

string

Interface: is a powerful feature that allows you to define a contract for an objects shape. It specifies the properties and their types that an object must have to be considered to the particular interface type.

Interfaces are primarily used for type-checking during development and do not generate any Javascript code at runtime.

interface Greet{

   name:string;

   age:number

}

const greets:Greet={

   name:'vinod',

   age:29

}

Watch Mode: tsc index.ts --watch 🡪for single file

For multiple file create tsconfig file then apply tsc --watch or tsc -w

RootDir and OutDir:

"rootDir": "./src",                                  /\* Specify the root folder within your source files. \*/

"outDir": "./dist",                                   /\* Specify an output folder for all emitted files. \*/

Object-Oriented Programming:

* A Class in terms of OOP is a blueprint for creating objects.
* A class is like a blueprint for creating similar things.

In Typescript, there is a convention to use Pascal Case (also known as Upper Camel Case) for class names.

//Class

class Persons{

    name:string ='Vinod';

    age: number =22;

    hobbies: string[] =["Reading","Painting"];

}

//Instance of a Class

const per :Persons =new Persons();

console.log(per);

class Persons{

    name: string;

    age: number;

    hobbies: string[];

    constructor(name:string,age:number, hobbies:string[]){

        this.name=name;

        this.age=age;

        this.hobbies=hobbies;

    }

}

//Instance of a Class

const person1 :Persons =new Persons("Vinod",25,["Reading","Painting"]);

const person2 :Persons =new Persons("Ram",25,["Cycling","Reading"]);

console.log(person1);

console.log(person2.hobbies);

Inheritance:

* It allows a class to reuse the functionality of an existing class without rewriting it.
* It is a mechanism in which one class acquires the property of another class. For Example, A child inherits the traits of his/her parents.

Super() keyword:

In Typescript, the super keyword is used in the context of class inheritance. It allows a subclass (also known as a derived class) to call methods or access properties of its superclass (also known as a base class). This is particularly useful when you want to extend the behaviour of a parent class while still leveraging its existing functionality.

class Persons{

    name: string;

    age: number;

    hobbies: string[];

    constructor(name:string,age:number, hobbies:string[]){

        this.name=name;

        this.age=age;

        this.hobbies=hobbies;

    }

    introduce():string{

        return `Hi, I'm ${this.name} and I'm ${this.age} year old. I have ${this.hobbies.join(',')}.`;

    }

}

class Student extends Persons{

     grade:number;

     constructor(name:string,age:number, hobbies:string[],grade:number){

        super(name,age,hobbies);

        this.grade=grade;

     }

     introduce():string{

        return `${super.introduce()} I am in grade ${this.grade}.`;

    }

}

//

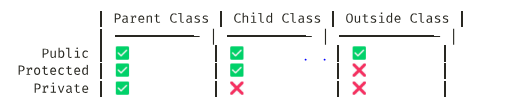
const person1 :Persons =new Persons("Vinod",25,["Reading","Painting"]);

const student1 :Student =new Student("Ram",25,["Cycling","Reading"],5);

console.log(person1.introduce());

console.log(student1.introduce());

Access Modifiers:



Shorthand Properties in Classes:

constructor(public name:string,public age:number, protected hobbies:string[]){

        }

Example:

class Persons{

        constructor(public name:string,public age:number, protected hobbies:string[]){

        }

        introduce():string{

            return `Hi, I'm ${this.name} and I'm ${this.age} year old. I have ${this.hobbies.join(',')}.`;

        }

    }

    class Student extends Persons{

         constructor(name:string,age:number, hobbies:string[], public grade:number){

            super(name,age,hobbies);

         }

         introduce():string{

            return `${super.introduce()} I am in grade ${this.grade}.`;

        }

    }

    //

    const person1 :Persons =new Persons("Vinod",25,["Reading","Painting"]);

    const student1 :Student =new Student("Ram",25,["Cycling","Reading"],5);

    console.log(person1.introduce());

    console.log(student1.introduce());

Getter and Setter Methods:

In typescript classes, you can use getter and setter methods to control the access and modification of class properties. Getter methods allow you to retrieve the value of a property, while setter methods allow you to set the value of a property with additional logic or validation.

class Persons{

       private \_age: number| undefined;

        constructor(public name:string,protected hobbies:string[]){

        }

        public set age(age:number){

            if(age>150 || age<0)

                throw new Error("Age is undefined");

            this.\_age=age;

        }

        public get age(){

            if(this.\_age===undefined)

                throw new Error("Age is undefined");

            return this.\_age;

        }

        introduce():string{

            return `Hi, I'm ${this.name} and I'm ${this.\_age} year old. I have ${this.hobbies.join(',')}.`;

        }

}

    //

    const person1 :Persons =new Persons("Vinod",["Reading","Painting"]);

    console.log(person1.introduce());

    person1.age=24;

    console.log(person1.age);

Static Properties and Methods:

Static methods and properties belong to the class itself rather than to instances of the class. By making methods and properties static, we can access them directly from the class without needing to create an instance of the class. This is useful for utility functions or properties that don’t rely on instance-specific data.

For example: Math operations Utility - creating a utility class to perform various mathematical operations.

class Mathoperations{

    public static PI:number=Math.PI;

    public static add(num1:number,num2:number){

        return num1+num2;

    }

}

console.log(Mathoperations.PI);

console.log(Mathoperations.add(45,45));

Abstract Classes:

* + - Abstract classes provide a way to define common properties and methods that multiple derived classes can share. This promotes code reuse and helps establish a common interface for related classes.
    - Abstract class cannot be instantiated.
* Abstract classes focus on class inheritance and sharing common functionality.
* Whereas the use context hook in React focuses on managing global state and allowing components to consume that state.

abstract class Shape{

    constructor(protected color:string){}

    abstract calculateArea():number;

    abstract displayArea:()=>void;

}

class Circle extends Shape{

    constructor(protected color:string,protected radius:number){

        super(color);

    }

    public calculateArea(): number {

        return Math.PI\*this.radius\*this.radius;

    }

    public displayArea= () => {

        console.log(`This is a ${this.color} circle with radius ${this.radius}.`)

    }

}

const circle=new Circle('red',5);

console.log(circle.calculateArea());

circle.displayArea();

Types vs. Interfaces:

1:

* Use custom types when you need unions, intersections, or mapped types.
* Use interfaces when defining object shapes or classes that adhere to a contract.

2:

* Interfaces can extend other interfaces to inherit their members.
* Custom types can use unions and intersections for more complex type compositions.

TYPE:

type Stud {

    name: string; age: number; };

type StudAddr = {

    city: string; state: string; };

type Data =Stud | StudAddr;

const BioData: Data = {

     name: "vinod",city: "Pune", state: "MH", };

console.log(BioData);

INTERFACE:

interface Stud {

    name: string;

    age: number;

}

interface StudAddr {

    city: string;

    state: string;

}

interface Data extends Stud, StudAddr {}

class BioData implements Data {

    constructor(

    public name: string,

    public age: number,

    public city: string,

    public state: string

    ){}

}

const std1= new BioData("vinod", 29, "pune", "MH");

console.log(BioData);

Type Safety with Typeof Guards:

A typeof guard in TypeScript lets you narrow down the type of a variable based on its runtime value. In TypeScript, type narrowing allows you to write type-safe code by ensuring you only operate on the correct type under certain circumstances. This is particularly useful with union types and generic types.

const favHobbies = (hobby) => {

    if (typeof hobby === "object" || Array.isArray(hobby)) {

        return hobby.map(() => { });

    }

    else {

        console.log(hobby);

    }

};

favHobbies("coding");

favHobbies(["coding", "games"]);