TypeScripts Documentation:-

TypeScript allows specifying the types of data being passed around within the code, and has the ability to report errors when the types don't match.

For example, TypeScript will report an error when passing a string into a function that expects a number. JavaScript will not.

A common way to use TypeScript is to use the official TypeScript compiler, which transpiles TypeScript code into JavaScript.

**Typescript installation command**

npm install typescript --save-dev

npx tsc –init //to initialise a json file with recommended settings

Working of type scripts:

// Greets the world.

console.log("Hello world!");

var x = 'Hi there';

x = 4;  //the type won't be converted again

While converting to js using tsc it will through an error what javascript may not give

npx tsc FirstProgram.ts

FirstProgram.ts:4:1 - error TS2322: Type 'number' is not assignable to type 'string'.

4 x = 4; //the type won't be converted again

~

Found 1 error in FirstProgram.ts:4

any is a type that disables type checking and effectively allows all types to be used.

let x = true;

x = 'string';

let v: any = true; //explicit type declaration

v = 'string';

console.log(typeof x);

console.log(typeof v);

FirstProgram.ts:2:1 - error TS2322: Type 'string' is not assignable to type 'boolean'.

2 x = 'string';

~

Found 1 error in FirstProgram.ts:2

Typescripts has special syntax for array

const x = [];

x.push('vivek');

The above code will work fine in typescript as well as in javascripts.

const x : string[] = [];

x.push('Vivek');

It will strictly check the type

const x : string[] = [];

x.push(1);

FirstProgram.ts:2:8 - error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.

The readonly keyword can prevent arrays from being changed.

const x : readonly string[] = [];

x.push('vivek');

FirstProgram.ts:2:3 - error TS2339: Property 'push' does not exist on type 'readonly string[]'.

A **tuple** is a typed array with a pre-defined length and types for each index.

Tuples are great because they allow each element in the array to be a known type of value.

let x : [number, boolean, string];

x = [1,true,'amit']

It will work correctly

let x : [number, boolean, string];

x = ['1',true,true]

FirstProgram.ts:2:6 - error TS2322: Type 'string' is not assignable to type 'number'.

FirstProgram.ts:2:15 - error TS2322: Type 'boolean' is not assignable to type 'string'.

// define our tuple

let x: [number, boolean, string];

// initialize correctly

x = [5, false, 'Hi there'];

// We have no type safety in our tuple for indexes 3+

x.push('Something new and wrong');

console.log(x);

But we mainly use readonly tuple here:

// define our tuple

let x:readonly [number, boolean, string];

// initialize correctly

x = [5, false, 'Hi there'];

// We have no type safety in our tuple for indexes 3+

x.push('Something new and wrong');

console.log(x);

FirstProgram.ts:6:3 - error TS2339: Property 'push' does not exist on type 'readonly [number, boolean, string]'.

**Object Declaration:**

const x : {name:string,age:number,eligible?:boolean}

= {

    name: 'Shovon Raul',

    age:22

}

//x.eligible=true;

console.log(x);

Here the eligible property of the runtime object x is option so I used this syntax there.

**Enum Declaration**

enum CardinalDirections {

    North = 1,

    East,

    South,

    West

  }

  // logs 1

  console.log(CardinalDirections.North);

  // logs 4

  console.log(CardinalDirections.West);

After changing it the corresponding javascript code will be

var CardinalDirections;

(function (CardinalDirections) {

    CardinalDirections[CardinalDirections["North"] = 1] = "North";

    CardinalDirections[CardinalDirections["East"] = 2] = "East";

    CardinalDirections[CardinalDirections["South"] = 3] = "South";

    CardinalDirections[CardinalDirections["West"] = 4] = "West";

})(CardinalDirections || (CardinalDirections = {}));

// logs 1

console.log(CardinalDirections.North);

// logs 4

console.log(CardinalDirections.West);

Type Aliases allow defining types with a custom name (an Alias).

Type Aliases can be used for primitives like string or more complex types such as objects and arrays:

type Name = String;

type Neumeric = Number;

type Eligibility = Boolean;

type Identity ={

    firstname: Name,

    lastName :Name,

    age : Neumeric,

    id: Neumeric,

    above18 ?: Eligibility,  //optional

    canVote : Eligibility

}

const person1:Identity ={

    firstname : 1,

    lastName : 'Raul',

    age: 22,

    id: 3033,

    canVote: true

};

FirstProgram.ts:14:5 - error TS2322: Type 'number' is not assignable to type 'String'.

14 firstname : 1,

~~~~~~~~~

FirstProgram.ts:5:5

5 firstname: Name,

~~~~~~~~~

The expected type comes from property 'firstname' which is declared here on type 'Identity'

type Name = String;

type Neumeric = Number;

type Eligibility = Boolean;

type Identity ={

    firstname: Name,

    lastName :Name,

    age : Neumeric,

    id: Neumeric,

    above18 ?: Eligibility,  //optional

    canVote : Eligibility

}

const person1:Identity ={

    firstname : 'Shovon',

    lastName : 'Raul',

    age: 22,

    id: 3033,

    canVote: true

};

**Corresponding Javascript created as:**

var person1 = {

    firstname: 'Shovon',

    lastName: 'Raul',

    age: 22,

    id: 3033,

    canVote: true

};

Interfaces are similar to type aliases, except they **only** apply to object types.

interface Identity {

    firstname: string,

    lastName :string,

    age : number,

    id: number,

    above18 ?: boolean,  //optional

    canVote : boolean

}

const person1:Identity ={

    firstname : 'Shovon',

    lastName : 'Raul',

    age: 22,

    id: 3033,

    canVote: true

};

interface Identity {

    firstname: string,

    lastName :string,

    age : number,

    id: number,

    above18 ?: boolean,  //optional

    canVote : boolean

}

interface Student extends Identity {

    rollNo : number;

    college : string

}

const person1:Identity ={

    firstname : 'Shovon',

    lastName : 'Raul',

    age: 22,

    id: 3033,

    canVote: true

};

const person2: Student = {

    firstname : 'Ajit',

    lastName: 'kumar',

    age: 22,

    id: 1000,

    canVote:true,

    rollNo: 2,

    college: 'Aditya College of Engineering and Technology'

}

We can extend the interface property as described above. The corresponding javascript file will be

var person1 = {

    firstname: 'Shovon',

    lastName: 'Raul',

    age: 22,

    id: 3033,

    canVote: true

};

var person2 = {

    firstname: 'Ajit',

    lastName: 'kumar',

    age: 22,

    id: 1000,

    canVote: true,

    rollNo: 2,

    college: 'Aditya College of Engineering and Technology'

};

Typescript union type: It is used when a particular variable required multiple datatypes.

Very efficiently we can use it in a function as followed.

function printStatusCode(code: string | number) {

    console.log(`My status code is ${code}.`)

  }

  printStatusCode(404);

  printStatusCode('name');

But we can not include string method into it in the function declaration.

function printStatusCode(code: string | number) {

    console.log(`My status code is ${code.toUpperCase()}.`)

  }

  printStatusCode(404);

  printStatusCode('name');

FirstProgram.ts:2:43 - error TS2339: Property 'toUpperCase' does not exist on type 'string | number'.

Property 'toUpperCase' does not exist on type 'number'.

TypeScript Function:

function multiply(a: number, b: number) {

    return a \* b;

  }

console.log(multiply(2,3));

Named Parameters

Typing named parameters follows the same pattern as typing normal parameters.

function divide({ dividend, divisor }: { dividend: number, divisor: number }) {

    return dividend / divisor;

  }

console.log(divide({dividend:9,divisor:3}));

Corresponding javascript file

function divide(\_a) {

    var dividend = \_a.dividend, divisor = \_a.divisor;

    return dividend / divisor;

}

console.log(divide({ dividend: 9, divisor: 3 }));

Rest parameters can be typed like normal parameters, but the type must be an array as rest parameters are always arrays.

function add(a: number, b: number, ...rest: number[]) {

    return a + b + rest.reduce((p, c) => p + c, 0);

  }

console.log(add(2,3,4,5,6,7,78))

Corresponding javascript file

function add(a, b) {

    var rest = [];

    for (var \_i = 2; \_i < arguments.length; \_i++) {

        rest[\_i - 2] = arguments[\_i];

    }

    return a + b + rest.reduce(function (p, c) { return p + c; }, 0);

}

console.log(add(2, 3, 4, 5, 6, 7, 78));

**TYPECASTING IN TYPESCRIPTS**

There are times when working with types where it's necessary to override the type of a variable, such as when incorrect types are provided by a library.

Casting is the process of overriding a type.

let x: unknown = 'hello';

console.log((x as string));

let y: unknown = 'hola'

console.log((<string>y).length);

**TypeScript Classes:-**

Member visibility:-

There are three main visibility modifiers in TypeScript.

public - (default) allows access to the class member from anywhere

private - only allows access to the class member from within the class

protected - allows access to the class member from itself and any classes that inherit it.

class Person {

    private name: string;

    public constructor(name: string) {

      this.name = name;

    }

    public getName(): string {

      return this.name;

    }

  }

  const person = new Person("Jane");

  console.log(person.getName());

Corresponding Javascript code

var Person = /\*\* @class \*/ (function () {

    function Person(name) {

        this.name = name;

    }

    Person.prototype.getName = function () {

        return this.name;

    };

    return Person;

}());

var person = new Person("Jane");

console.log(person.getName());

If the function is private in the typescript it will through an error while calling the function. We can include readonly properties for variables also.

CLASS INHERITANCE CAN BE IMPLEMENTED AS FOLLOWED

Classes can extend each other through the extends keyword. A class can only extends one other class.

interface Shape {

    getArea: () => number;

  }

  class Rectangle implements Shape {

    public constructor(protected readonly width: number, protected readonly height: number) {}

    public getArea(): number {

      return this.width \* this.height;

    }

  }

  class Square extends Rectangle {

    public constructor(width: number) {

      super(width, width);

    }

    // getArea gets inherited from Rectangle

  }

  class Cube extends Square{

    public override getArea(): number {

        return this.width\*this.width\*this.width;

    }

  }

let room = new Cube(4);

console.log(room.getArea());

When a class extends another class, it can replace the members of the parent class with the same name.

Newer versions of TypeScript allow explicitly marking this with the override keyword.

**ABSTRUCT CLASS:**

Classes can be written in a way that allows them to be used as a base class for other classes without having to implement all the members. This is done by using the abstract keyword. Members that are left unimplemented also use the abstract keyword.

abstract class Polygon {

    public abstract getArea(): number;

    public toString(): string {

      return `Polygon[area = ${this.getArea()}]`;

    }

  }

  class Rectangle extends Polygon {

    public constructor(protected readonly width: number, protected readonly height: number) {

      super();

    }

    public getArea(): number {

      return this.width \* this.height;

    }

  }

let home = new Rectangle(2,3);

console.log(home.getArea());

console.log(home.toString());

**TYPESCRIPTS BASIC GENERICS:**

Generics allow creating 'type variables' which can be used to create classes, functions & type aliases that don't need to explicitly define the types that they use.

Generics makes it easier to write reusable code.

function createPair<T1,T2>(V1:T1,V2:T2):[T1,T2]{

    return [V1,V2];

}

console.log(createPair<string,number>("Shovon Raul",3033))

Generics for classes:

class NamedValue<T> {

    private \_value: T | undefined;

    constructor(private name: string) {}

    public setValue(value: T) {

      this.\_value = value;

    }

    public getValue(): T | undefined {

      return this.\_value;

    }

    public toString(): string {

      return `${this.name}: ${this.\_value}`;

    }

  }

  let value = new NamedValue<number>('myNumber');

  value.setValue(10);

  console.log(value.toString()); // myNumber: 10

  value.setValue(3.2345677);

  console.log(value.toString());