**TYPESCRIPT**

**Purpose** : It identifies type-errors during compilation to JavaScript and helps in fixing potential bugs during runtime.

Ex: let name = “Vinayak”;

console.log(“age >>”, name.age) 🡺 While writing this line of code, It will show this error message.

const personName:string = 'Vinayak';

const personAge:number = 15;

const isPersonHealthy = true;

const players:string[] = ["adam", "pointing", "kohli"];

const scores:number[] = [100, 200, 300];

Note : If type is string means it can store any string data. But here I am strictly saying accept only these two string values

type Result = "success" | "failure";

const result:Result = "success";

type CelebrationScore = 50 | 100;

const celebrationAt:CelebrationScore = 100;

Note : **Interface** shows you what are the properties and what are the types of those properties of the object.

interface Employee {

    name : string | number,

    company : string,

    year : number,

    currentlyWorking : true

}

const obj:Employee = {

    name : "Girish",

    company : "Altair",

    year : 2000,

    currentlyWorking : true

}

// codevolution – test this at 17 mins

let nameOfThePerson:null = null;

let age:undefined = undefined;

let result:string = undefined;

**TUPLES** : contain elements of different types but the sequence, type & the number of elements should match with the type.

let arr1:[string, number] = ['vinayak', 15];

**ENUMS** : assigns a number in the ascending order by default or u can assign your own number to our values like red, green, yellow.

enum Color { Red, Green, Blue}

let color:Color = Color.Blue;

Note : Internally which is equal to let Color = { Red : 0, Green : 1, Blue : 2};

enum ColorTwo { Red = 5, Green = 60, Blue}

let color: ColorTwo = ColorTwo.Blue;

Note : Internally which is equal to let ColorTwo = { Red : 5, Green : 60, Blue :61};

And you can not access like, let color : ColorTwo = Color.Yellow => wrong

**Type : any**

let x:any = 'vinu';

x = 15;

x.toUpperCase()

x.toFixed()

Note : It ***wont throw*** any error now coz type is any.

**Type : unknown**

let x:unknown = 'vinu';

x = 15;

x = true;

x.name = "vinayak sonar";

x.getName = function(){ return this.name };

x.name = "hi";

x.toUpperCase()

x.toFixed()

Note : It will ***throw error*** saying u can not assign any properties or any methods to the type unknown.

U can store any value in unknown type including objects, but u can’t access properties and methods.

let num : unknown = 'red';

(num as string).toUpperCase()

**Type : Multitype**

let x : number | string;

x = 15;

x = "Hello";

x.toUpperCase();

**FUNCTIONS**

function sum(a:number, b?:number):number {

    return b ? a + b : a

}

sum(10, 5)

sum(10)

default parameters of a function.

function totalSum(a:number, b:number = 100):number {

    return a + b;

}

console.log(totalSum(10))

**INTERFACE**

interface Person {

    firstName : string,

    age : number,

    engaged ?: true

}

Note : p2 object, if it has no engaged value then its fine. No error.

let p1:Person = { firstName : "Vinayak", age : 15, engaged : true }

let p2:Person = { firstName : "Soumya", age : 10, }

function personDetails(obj : Person){

    return `${obj.firstName} is of age ${obj.age}`

}

let output = personDetails(p2);

console.log(output)

"Soumya is of age 10"

function personDetailsDefault(obj : Person = { firstName : "hi", age : 15}){

    return `${obj.firstName} is of age ${obj.age}`

}

let outputTwo = personDetailsDefault();

console.log(outputTwo)

"hi is of age 15"

**DIFFERENCE BETWEEN ANY and UNKNOWN**

Note : If type is **any**, u can access properties and methods using . operator, in which you don’t see any error

If type is **unknown**, u see errors instantly. To access properties u have to say ***treat*** the variable as a string then apply toUpperCase() method.

( \*\*\* I wont convert to number to string or string to number )

let x: unknown = 'hi';

x = "hi vinayak";

let upperX = (x as string).toUpperCase();

console.log(upperX);

let y: unknown = '16';

y = 13.456664567457;

let numbFixedTo = (y as number).toFixed(2);

console.log(numbFixedTo);

**CLASSES**

class Employee {

    empName : string;

    constructor(nm : string){

        this.empName = nm;

    }

    get name(){

        return this.empName

    }

    greeting(){

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

    }

}

const emp1 = new Employee('Vinayak');

emp1.greeting()

class Manager extends Employee {

    specialChair : boolean;

    constructor(managerName : string){

        super(managerName);

        this.specialChair = false;

    }

    set setSpecialChair(flag : boolean){

        this.specialChair =  flag;

    }

    printAssets(){

        console.log(`${this.empName} has a spl chair ? ${this.specialChair} `)

    }

}

const managerOne = new Manager('Shreenivasa');

managerOne.specialChair = false;

managerOne.setSpecialChair = true;

managerOne.printAssets()

**ACCESS MODIFIERS**

**Private** :

Properties and methods inside the class, are accessible, only by being inside class. (this.property and this.methodName)

Not accessible by methods in the derived class, also.

**Public**:

By default. Whenever u create an instance using new keyword, u can access properties and methods.

**Protected** :

Accessible by derived class. But not by outside meaning, when u create an instance using new keyword.

class Employee {

    protected empName : string;

    private salary : number;

    constructor(nm : string){

        this.empName = nm;

        this.salary = 0;

    }

    // Getter

    get name(){

        return this.empName

    }

    // setter

    // I am protected and can be called only by derived class

    protected set setEmpSalary(sal : number){

        this.salary = sal;

    }

    protected get printSalary(){

        return this.salary;

    }

    // I am public and any one can call me.

    public greeting(){

        console.log(`Employee : Hi Good Morning ${this.empName}`)

    }

}

const emp1 = new Employee('Vinayak');

emp1.greeting()

class Manager extends Employee {

    public specialChair : boolean;

    constructor(managerName : string){

        super(managerName);

        this.specialChair = false;

    }

    public set setSpecialChair(flag : boolean){

        this.specialChair =  flag;

    }

    public printAssets(){

        console.log(`${this.empName} has a spl chair ? ${this.specialChair} `)

    }

    public setManagerSalary(sal :number){

        // setter in the parent class

        this.setEmpSalary = sal;

    }

    public printManagerSalary(){

        console.log(`Manager salary is ${this.printSalary}`)

    }

}

const managerOne = new Manager('Shreenivasa');

managerOne.greeting();

managerOne.specialChair = false;

managerOne.setSpecialChair = true;

managerOne.printAssets()

managerOne.setManagerSalary(50000)

managerOne.printManagerSalary()

**TYPESCRIPT IN REACT**