assertion & type narrowing

যখন টাইপস্ক্রিপ্টের আগেই আমরা প্রোগ্রামাররা একটা ভ্যারিয়াবেলের টাইপ বুঝতে পারি, এবং সেটা as লিখে assumption করে দেই, তখন তাকে type assertion বলে । যেমনঃ

let carDetails : any ;

carDetails = 'I only got 6 cars' ;

(carDetails as string).concat ; //-- we know the variable will be string.

carDetails = 2500 ;

(carDetails as number); //-- we know the variable will be number

For ফাংশানঃ

const kgToGm = (value : number | string) : string | number | undefined => {

if(typeof value === 'number'){

return value \* 1000 ;

}

if(typeof value === 'string'){

const convertedValue = parseInt(value) \* 1000

return `Summation is ${convertedValue}`

}

}

const result1 = kgToGm(25) as number //-- we used 'as number' because we know the return will be number

const result2 = kgToGm('12') as string //-- we used 'as string' cz we know the return will be string

for ‘try catch’ block :

type customError = {

message : string ;

}

try{

}catch(error){

console.log((error as customError).message);

}

Interface, type vs interface

Type alias এর মতই আরেকটা সিস্টেম হচ্ছে type interface, যেমনঃ

**// --- type alias**

**type user = {**

**name : string;**

**profession : string;**

**salary ? : number;**

**married : boolean;**

**}**

**const rasel : user = {**

**name : 'rasel',**

**profession : 'webdev',**

**married : true**

**}**

**// --- interface**

**interface userWithInterface {**

**name : string;**

**profession : string;**

**salary ? : number;**

**married : boolean;**

**}**

**const abir : userWithInterface = {**

**name : 'rasel',**

**profession : 'webdev',**

**married : true**

**}**

Type alias এ যেমন একটি টাইপের সাথে আরেকটি টাইপ এড করা যায় , যেটাকে intersection বলে, তেমনি interface এর সাথেও intersection করা যায়। এবং এটা করতে হয় extends ব্যবহার করে।

**// --- 'intersection' system in type alias**

**type userWithId = user & {id : number}**

**const rasel2 : userWithId = {**

**name : 'rasel',**

**profession : 'webdev',**

**married : true,**

**id : 6663719**

**}**

**// --- 'intersection' system in interface**

**interface userWithInterface2 extends userWithInterface {id : number} ;**

**const abir2 : userWithInterface2 = {**

**name : 'abir',**

**profession : 'banker',**

**married : true,**

**id : 2,**

**}**

Interface for **array[ ]** :

**// --- interface for array[]**

**type roll = number[]; //-- type alias**

**const class6 : roll = [1,2,3,4,5]; //-- array with type alias**

**interface rollWithInterface {**

**[index : number] : number**

**}**

**const class7 : rollWithInterface = [1,2,3,4,5] //-- array with interface**

Interface for **function( )** :

**// --- interface for function**

**type add = (num : number, num2 : number) => number ; //-- type alias**

**const addNumber : add = (num1, num2) => num1 + num2 ; //-- function with type alias**

**interface addInterface {**

**(num : number, num2:number) : number //-- interface for function**

**}**

**const addNumber2 : addInterface = (num1, num2) => num1 + num2 ; //-- function with interface**

Generic Type

Generics হচ্ছে dynamically type ডিক্লেয়ার করার সিস্টেম। অর্থাৎ আমাদেরকে প্রথমেই টাইপ ডিক্লেয়ার করার প্রয়োজন নেই। ভ্যারিয়েবল ডিক্লেয়ার করার সময় আমরা টাইপ ডিফাইন/সেট করে দিতে পারবো।

const numbers : number[] = [2,3,4,5,6] ;

const numbers2 : Array<number> = [2,3,4,5,6] ; //-- generic type

const cars : string[] = ['audi', 'bmw', 'mercedez'] ;

const cars2 : Array<string> = ['audi', 'bmw', 'mercedez'] ; //-- generic type

const boolsArr : boolean[] = [true, false, true];

const boolsArr2 : Array<boolean> = [true, false, true]; //-- generic type

or we can use generic type dynamically like this :

type genericArray<T> = Array<T> ;

const numbers3 : genericArray<number> = [2,3,4,5,6] //-- dynamic generic type

const user : GenericArray<{name : string, city : string}> = [

{

name : 'rasel',

city : 'barisal'

},

{

name : 'jonathan',

city : 'dhaka',

}

]

Generic type tuple :

tuple কি ? আমরা যখন টাইপ স্ক্রিপ্ট দিয়ে কোন একটি এরে লিখবো তখন আমাদেরকে ওই এরের একটি মাত্র টাইপ ডিক্লেয়ার করতে হয় এবং ডিক্লেয়ার করা ওই টাইপই ব্যবহার করা যায়। যেমন ,

const cars4 : string[] = ['audi', 'bmw', 'mercedez'] ;

এই এরেটিতে যেহেতু টাইপ string[ ] ব্যবহার করেছি , তাই এখানে string টাইপ ডাটাই এসাইন করা যাবে। অন্য টাইপ ডাটা এসাইন করলে এরর আসবে। এখন অন্য টাইপ ডাটা ব্যবহার করতে হলে আমাদের tuple ইউজ করতে হবে। উদাহরণ,

const cars6 : [string, number, boolean] = ['bmw', 45, false] ;

অথবা

type carsTuple = [string, number, boolean] ;

const cars5:carsTuple = ['audi', 35, true] ;

generic type এর ক্ষেত্রেও tuple ব্যবহার কর যাবে।

type genericTuple <X, Y> = [X, Y] ;

const cars7 : genericTuple<string, number> = ['audi', 3];

const cars8 : genericTuple<string, {year : number, category : string}> = ['audi', {year : 2015, category : 'A'}]

Generic Interface

interface genericInterFace<T, X = null> {

name : string,

profession : string,

salary ? : number,

married : boolean,

cars : T ,

bikes ?: X

}

interface forAudi {

brand : string;

year : number;

color : string;

cc : number

}

interface forBike {

brand : string,

cc : number,

premium : boolean

}

const user : genericInterFace<forAudi, forBike> = {

name : 'rasel',

profession : 'dev',

married : true,

cars : {

brand : 'audi',

color : 'white',

year : 2015,

cc : 2200

},

bikes : {

brand : 'Yamaha',

cc : 150,

premium : true

}

}

const user2 : genericInterFace<forAudi> = {

name : 'rasel',

profession : 'dev',

married : true,

cars : {

brand : 'audi',

color : 'white',

year : 2015,

cc : 2200

}

}

Generic Functions

type normalFn = (num1 : number, num2 : number) => number ;

const normalFN : normalFn = (value1, value2) => value1+value2 ;

const regularFn = (value : string) : string[] => [value] ;

const regularResult = regularFn('Bangladesh') ;

const dummy = <T>(value : T) : T[] => [value] ;

const result = dummy<string>('Bangladesh');

const result2 = dummy<object>({name:'rasel', cars : false});

//-- we can write result2 like this

const result3 = dummy<{name:string, cars:boolean}>({name:'rasel', cars : false});

//-- we can write result3 like this

type user = {name:string, cars:boolean};

const result4 = dummy<user>({name:'rasel', cars : false});

//-- we can use tuple with generic functions

const dummy2 = <X, Y>(value1:X, value2:Y):[X, Y] => [value1, value2];

const resul5 = dummy2<number, string>(250, 'audi');

const addStudentToCourse = <X>(student:X) => {

const course = 'Learn Typescript' ;

return {...student, course} ;

}

const result6 = addStudentToCourse({name:'rasel', cars:false, home:true});

const result7 = addStudentToCourse<{name:string,cars:boolean, profession:string}>({name:'jonathan', cars:true, profession : 'assasin'});

constrain

generic দিয়ে আমরা যখন ডায়নামিক্যালি টাইপ সেট করে দিতে পারি, আমরা চাইলে ওই ডায়নামিক টাইপের সাথে আগে থেকে ঠিক করা যেকোন টাইপও পাঠাতে পারি। মানে ডায়নামিক্যালি টাইপ দেয়ার পাশাপাশি ইউজারকে জোর করে (enforce) কিছু টাইপ লিখতে বাধ্য করাতে পারি। এবং এটা করতে হবে extends কি-ওয়ার্ড দিয়ে,

const addStudentToCourse = <X extends {id:number, email:string, name:string}>(student:X) => {

const course = 'Learn Typescript' ;

return {...student, course} ;

}

const result = addStudentToCourse({name:'rasel', email:'rasel@gmail',id:3, cars:false, home:true});

const result2 = addStudentToCourse({name:'jonathan', email:'rasel@gmail', id:4, cars:true, profession : 'assasin'});

keyOf

type user = {

name : string;

id : number,

email : string

}

type userKey = 'name' | 'id' | 'email' ; //-- this is called keyOf

type userKey2 = keyof userKey ; //-- this is as same as the previous line

const rasel = {

name : 'rasel',

id : 35,

email : 'rasel@gmail'

}

const findUser = <X, Y extends keyof X>(obj:X, key:Y) => {

return obj[key]; //-- rasel[id] = 35 ;

}

const result = findUser(rasel, 'email');

asynchronous typescript

type todo = {

userId : number,

id : number,

title : string ,

completed : boolean

}

const showData = async () : Promise<todo> => {

const response = await fetch('https://jsonplaceholder.typicode.com/todos/1');

const data = await response.json();

console.log(data);

return data;

}

showData();

mapped

অন্য একটি type এর key গুলো দিয়ে আরেকটি টাইপ তৈরি করার জন্য mapped ইউজ করা হয়। এই mapped ইউজ করার সময় আমরা বলে দিতে পারি নতুন key গুলোর টাইপ কি হবে।

type cars = {

brand : string;

country : string;

sports : boolean

}

type keyOfCars = {

[index in keyof cars] : number ;

}

/\* output of keyOfCars :

type keyOfCars = {

brand: number;

country: number;

sports: number;

}

\*/

এখন উপরের এই কোডটাকে যদি আরও ডায়নামিক করতে চাই মানে , [index in keyof cars] : number ; মানে এখানে যেহেতু number টাইপ সেট করে দিয়েছি, তাহলে সবগুলা key-তেই আমাদের নাম্বার টাইপ ডাটা সেট করতে হবে। কিন্ত আমরা যদি এটা ডায়নামিক্যালি সেট করতে চাই অর্থাৎ যেখান থেকে কপি করেছি সেখানের মত হুবুহু টাইপ দিতে চাই তাহলে নিচের মত লেখা যায়।

type KeyOfCars2<T> = {

[index in keyof T] : T[index] ;

}

const findCars : KeyOfCars2<cars> = {

brand : 'audi',

country : 'england',

sports : false

}

utiltiy

type Person = {

name: string;

age: number;

email: string;

contactNo: string

}

type Result = Pick<Person, 'name' | 'age'>; //-- this 'Pick' utility will pick only 'name' & 'age' from Person type

type Result2 = Omit<Person, 'email' | 'name'>; //-- this 'Omit' utility will pick everything from type 'Person' except 'name' & 'email'

type Result3 = Required<Person> ; //-- this 'Required' utility will pick everything from Person type & it will make them required

type Result4 = Partial<Person>; //-- this 'Partial' utility will pick everything from Person type & it will make them optional

type PersonReadOnly = Readonly<Person> ;

const user : PersonReadOnly = {

name: 'rasel',

age: 24,

email: 'rasel@gmail',

contactNo: 'barisal'

}

user.name = 'jonathan' ; //-- it will show error as we made it read only types

type MyObj = Record<string, number> ; //-- with this utility we can define type of object all at once. i mean all the property type can be set for once

const obj1 : MyObj = {

name : 22,

email : 34

}

type MyObj2 = Record<string, unknown> ;

const obj2 : MyObj2 = {

name : 'rasel',

email : 34

}

//-- with this utility 'Record<string, unknown>' we set the property type unknown so that in the time usage we will be able use any type for property that we couldn't do for 'Record<string, number>' utility

class

javascript এ যেভাবে ক্লাস লেখেঃ

// ---- class in regular javascript

class Cars {

constructor(brand, country, year){

this.brand = brand;

this.country = country;

this.year = year ;

}

}

const audi = new Cars('audi','england',2015);

const bmw = new Cars('bmw','england',2022);

console.log(audi.year); // output = 2015

console.log(bmw.year); // output = 2022

audi.showDetails(); //-- output = 'I have bought audi from england in 2015' bmw.showDetails(); //-- output = 'I have bought bmw from england in 2022'

typescript এ যেভাবে ক্লাস লেখেঃ

// ---- class in typescript

class Cars {

brand: string;

country: string;

year: string;

constructor(brand: string, country: string, year: string) {

this.brand = brand;

this.country = country;

this.year = year;

}

//-- method

showDetails() {

console.log(`I have bought ${this.brand} from ${this.country} in ${this.year}`);

}

}

const audi = new Cars('audi', 'england', '2015');

const bmw = new Cars('bmw', 'england', '2022');

console.log(audi.year); // output = '2015'

console.log(bmw.year); // output = '2022'

audi.showDetails(); //-- output = 'I have bought audi from england in 2015'

উপরের সেইম কোডটি আমরা parameters ব্যবহার করে অনেক কম কোড লিখতে পারি।

// ---- class in typescript with paramters

class Cars {

// using 'parameters' in constructor

constructor(public brand: string, public country: string, public year: string) {}

//-- method

showDetails() {

console.log(`I have bought ${this.brand} from ${this.country} in ${this.year}`);

}

}

const audi = new Cars('audi', 'england', '2015');

const bmw = new Cars('bmw', 'england', '2022');

console.log(audi.year); // output = '2015'

console.log(bmw.year); // output = '2022'

audi.showDetails(); //-- output = 'I have bought audi from england in 2015'

bmw.showDetails(); //-- output = 'I have bought bmw from england in 2022'

inheritance

normal class in typescript :



উপরের এই ক্লাস যদি inheritence দিয়ে লেখিঃ



type guard : typeof

// --- 'typeOf' typeguard

type Alphaneumeric = number | string ;

const addNum = (number1 : Alphaneumeric, number2 : Alphaneumeric) : Alphaneumeric => {

if(typeof number1 === 'number' && typeof number2 === 'number'){

return number1 + number2;

}else{

return number1.toString() + number2.toString();

}

}

console.log(addNum(5,23)); // 28

console.log(addNum(5,'23')); // 523

type guard : ‘in’ guard

// --- 'in' typeguard

type NormalUser = {

name : string

}

type AdminUser = {

name : string;

role : string;

}

const getUser = (user : NormalUser | AdminUser) => {

if('role' in user){

console.log(`My name is ${user.name}, my role is ${user.role}`);

}else{

console.log(`My name is ${user.name}`);

}

}

const rasel : NormalUser = {

name : 'rasel',

}

const jonathan : AdminUser = {

name : 'jonathan',

role : 'admin'

}

getUser(rasel); // My name is rasel

getUser(jonathan); // My name is jonathan, my role is admin

type guard : ‘ instanceOf ’

class Car {

brand : string;

color : string;

country : string;

constructor(brand:string, color:string, country:string){

this.brand = brand ;

this.color = color ;

this.country = country ;

}

myCar(){

console.log(`${this.brand} is my car from ${this.country} in ${this.color} color`);

}

}

class Audi extends Car {

constructor(brand:string, color:string, country:string){

super(brand, color, country) ;

}

featureForAudi(){

console.log(`${this.brand} has six rings`);

}

}

class Mercedez extends Car {

constructor(brand:string, color:string, country:string){

super(brand, color, country) ;

}

featureForMercedez(){

console.log(`${this.brand} is named after founder's daughter`);

}

}

const getCar = (car : Car) => {

if(car instanceof Audi){

car.featureForAudi();

}else if(car instanceof Mercedez){

car.featureForMercedez();

}else{

car.myCar();

}

}

const myCar1 = new Audi('audi', 'white', 'england');

const myCar2 = new Mercedez('mercedez', 'silver', 'canada');

getCar(myCar1); // audi has six rings

getCar(myCar2); // mercedez is named after founder's daughter

getCar(new Car('toyota', 'white', 'japan')) // toyota is my car from japan in white color

//-- or we can write this code in this way

{

class Car {

brand : string;

color : string;

country : string;

constructor(brand:string, color:string, country:string){

this.brand = brand ;

this.color = color ;

this.country = country ;

}

myCar(){

console.log(`${this.brand} is my car from ${this.country} in ${this.color} color`);

}

}

class Audi extends Car {

constructor(brand:string, color:string, country:string){

super(brand, color, country) ;

}

featureForAudi(){

console.log(`${this.brand} has six rings`);

}

}

class Mercedez extends Car {

constructor(brand:string, color:string, country:string){

super(brand, color, country) ;

}

featureForMercedez(){

console.log(`${this.brand} is named after founder's daughter`);

}

}

const isAudi = (car : Car) : car is Audi => {

return car instanceof Audi;

}

const isMercedez = (car : Car) : car is Mercedez => {

return car instanceof Mercedez;

}

const getCar = (car : Car) => {

if(isAudi(car)){

car.featureForAudi();

}else if(isMercedez(car)){

car.featureForMercedez();

}else{

car.myCar();

}

}

const myCar1 = new Audi('audi', 'white', 'england');

const myCar2 = new Mercedez('mercedez', 'silver', 'canada');

getCar(myCar1); // audi has six rings

getCar(myCar2); // mercedez is named after founder's daughter

getCar(new Car('toyota', 'white', 'japan')) // toyota is my car from japan in white color

}

type guard : ‘ access modifiers = public, private, protected ’

{

// --- Access Modiefiers.

// --- Public, Private, Protected

class BankAccount {

public id: number;

public name: string;

public balance: number; //-- we can change this value anytime from anywhere

constructor(id: number, name: string, balance: number) {

this.id = id;

this.name = name;

this.balance = balance;

}

}

const myAccount = new BankAccount(22, 'rasel', 2500);

myAccount.balance = 200; //-- we can change any property value if it's public

console.log(myAccount.balance); // 200

}

{

// --- Access Modiefiers.

// --- Public, Private, Protected

class BankAccount {

public id: number;

public name: string;

private \_balance: number; //-- we cannot change this property value cz it's 'private'. we have to add a method to change private property

constructor(id: number, name: string, \_balance: number) {

this.id = id;

this.name = name;

this.\_balance = \_balance;

}

// -- we have to create method for changing private property value. in this case we will change 'balance' property which is private

addBalance = (amount : number) => {

this.\_balance = this.\_balance + amount ;

}

getbalance = () : number => {

return this.\_balance;

}

}

const myAccount = new BankAccount(22, 'rasel', 2500);

// myAccount.balance = 200; //-- we cannot change property value if it's 'private'

myAccount.addBalance(5000) ;

console.log(myAccount.getbalance()); // 7500

}

কিন্ত কোন প্রপার্টি প্রাইভেট করা থাকলে , সেটার class দিয়ে যদি আরেকটা সাব-ক্লাস (derived class) তৈরি করি, তাহলে সেই সাব-ক্লাসের ভেতর উক্ত private প্রোপার্টি এক্সেস করতে পারবোনা। তাই উক্ত প্রোপার্টি এক্সেস করতে হলে একই সাথে সেটাতে প্রাইভেট বৈশিষ্ট্য বজায় রাখতে চাইলে , আমরা সেখানে private এর বদলে protected ব্যবহার করবো।

{

// --- Access Modiefiers.

// --- Public, Private, Protected

class BankAccount {

public id: number;

public name: string;

protected balance: number; //-- we can change this value anytime from anywhere

constructor(id: number, name: string, balance: number) {

this.id = id;

this.name = name;

this.balance = balance;

}

}

// const myAccount = new BankAccount(22, 'rasel', 2500);

class StudentAccount extends BankAccount {

test(){

this.balance = 999 ; //-- we would not able to see 'balance' property if it was 'private' in parent property

console.log(this.balance);

}

}

}

Getter & setter method

// --- getter & setter method

class BankAccount {

public id: number;

public name: string;

private \_balance: number; //-- we cannot change this property value cz it's 'private'. we have to add a method to change private property

constructor(id: number, name: string, \_balance: number) {

this.id = id;

this.name = name;

this.\_balance = \_balance;

}

// --- normal method

getbalance = () : number => {

console.log(this.\_balance);

return this.\_balance;

}

// --- same method with 'getter'

get getBalance2(){

console.log(this.\_balance);

return this.\_balance;

}

// -- normal method

addBalance = (amount : number) => {

this.\_balance = this.\_balance + amount ;

}

// --- same method with 'setter'

set addBalance2(amount : number){

this.\_balance = this.\_balance + amount ;

}

}

const studentAccount = new BankAccount(55, 'rasel', 3500);

//--- set value by normal method

studentAccount.addBalance(500);

//--- set value by setter method

studentAccount.addBalance2 = 500;

//--- get value by normal method

studentAccount.getbalance();

//--- get value by getter method

studentAccount.getBalance2;

statics in oop

{

class Counter {

count: number = 0;

increamentCount() {

return this.count = this.count + 1;

}

decreamentCount() {

return this.count = this.count - 1;

}

}

const counter1 = new Counter();

const counter2 = new Counter();

console.log(counter1.increamentCount());;

console.log(counter2.increamentCount());;

}

// --- now if we do this with static

{

class Counter {

static count: number = 0;

increamentCount() {

return Counter.count = Counter.count + 1;

}

decreamentCount() {

return Counter.count = Counter.count - 1;

}

}

const counter1 = new Counter();

const counter2 = new Counter();

console.log(counter1.increamentCount()); // 1

console.log(counter2.increamentCount()); // 2

}

{

class Counter {

static count: number = 0;

static increamentCount() {

return Counter.count = Counter.count + 1;

}

static decreamentCount() {

return Counter.count = Counter.count - 1;

}

}

const counter1 = new Counter();

const counter2 = new Counter();

console.log(Counter.increamentCount()); // 1

console.log(Counter.increamentCount()); // 2

}

Polymorphism

কোন প্যারেন্ট ক্লাসের সেইম মেথড যখন তার child অথবা derived ক্লাস ব্যবহার করে কিন্তু ভিন্ন আউটপুট দেয় তখন সেটাকে polimorphism বলে।

class Person {

getName(){

return 'My name is X' ;

}

}

class Rasel extends Person{

getName(): string {

return 'My name is Rasel';

}

}

class Jonathan extends Person {

getName(): string {

return 'My name is Jonathan';

}

}

const person1 = new Rasel();

const person2 = new Jonathan();

const showNames = (person : Person) => {

console.log(person.getName());

}

showNames(person1); // My name is Rasel

showNames(person2); // My name is Jonathan

// --- different example

class Cars {

getCarDetails(){

return '';

}

}

class Audi extends Cars {

color : string ;

constructor(color:string){

super();

this.color = color ;

}

getCarDetails(): string {

return `I have Audi which is in ${this.color} `

}

}

class Mercedez extends Cars {

brand : string ;

country : string ;

constructor(brand:string, country:string){

super();

this.brand = brand ;

this.country = country ;

}

getCarDetails(): string {

return `${this.brand} is my car from ${this.country}` ;

}

}

const car1 = new Audi('white');

const car2 = new Mercedez('Mercedez', 'Canada');

const getDetails = (param : Cars) => {

console.log(param.getCarDetails());

}

getDetails(car1); // I have Audi which is in white

getDetails(car2); // Mercedez is my car from Canada

Interview Question

1.  **What are some benefits of using TypeScript over JavaScript in a project ?**

Some of the benefits of using Typescript over Javascript are written below in points

* TypeScript saves development time and allows developers to become more efficient.
* TypeScript offers variable declaration, functional paradigm, and type system, which JavaScript doesn’t offer.
* Since Microsoft backs TypeScript, it has many leading frameworks and editors. Through tight integration with editors, it provides error handling during compilation to avoid errors at runtime.
* If you have a large and complex codebase, there are higher chances of errors occurring. However, that would be good if some errors were resolved during the compilation time. This is when you can use TypeScript to reduce errors during compile time.
* TypeScript runs seamlessly for large projects or when many developers are working together.
* As the codebase grows, TypeScript code is simpler to debug since type errors can be found during compilation rather than during runtime.

1.  **What is the purpose of the optional chaining (?.) and nullish coalescing (??) operators in TypeScript, and how do they work? Provide an example for each**

When we declare types in typescript , sometimes we need some type to be optional. Like if the user doesn’t use that type later , it shouldn’t show error. To make a type optional we can use optional chaining (?) like below :

type userType = {

name : string;

address : {

city : string ;

state : string ;

permanentAddress ? : string ;

}

}

const user : userType = {

name : 'rasel',

address : {

city : 'bhandaria',

state : 'barisal',

}

}

In the above we declared a type permanentAddress ? : string but we haven’t defined it while declaring variable below named const user : userType and didn’t get a error. If we wouldn’t use (?) optional chaining there would be an error.

Purpose of using nullish coalescing operator : if we want to get a result depending on the value is null or undefined only, we can use use nullish coalescing (??) operator.

const isAudi = null ;

const result1 = isAudi ?? 'My Car is Audi' ;

console.log(result1); //output: My Car is Audi

এখানে isAudi = null বিধায় My car is audi প্রিন্ট হয়েছে। যদি null বা undefined না হয়ে অন্য ভ্যালু হতো তাহলে সেই অন্য ভ্যালু প্রিন্ট হতো। যেমনঃ

const isAudi = 'Bmw' ;

const result1 = isAudi ?? 'My Car is Audi' ;

console.log(result1); //output: Bmw

3. **How do you handle asynchronous operations in TypeScript, and what are the advantages of using async/await over callbacks or Promises?.**

In typescript we can handle asynchronous operations using callbacks, Promisess or async/await method. Here is an example of asynchronous operations using async/await in typescript :

type todo = {

userId : number,

id : number,

title : string ,

completed : boolean

}

const showData = async () : Promise<todo> => {

const response = await fetch('https://jsonplaceholder.typicode.com/todos/1');

const data = await response.json();

console.log(data);

return data;

}

showData();

There are some advantages of using async/await over callbacks or Promises :

* async/await makes asynchronous code look more like synchronous code, improving readability and reducing callback hell.
* We can use try/catch with async/await . Such way we can handle error more cleaner way.
* async/await is built on top of Promises, so it integrates well with existing Promise-based code.

5. **Explain the role of type guards in TypeScript and provide an example of a custom type guard.**

In typescript, if we want to show different result based on the type we get from variable during runtime, we can use typeguard typeof or typeIn. Here is an example of type guard (typeOf):

// --- 'typeOf' typeguard

type Alphaneumeric = number | string ;

const addNum = (number1 : Alphaneumeric, number2 : Alphaneumeric) : Alphaneumeric => {

if(typeof number1 === 'number' && typeof number2 === 'number'){

return number1 + number2;

}else{

return number1.toString() + number2.toString();

}

}

console.log(addNum(5,23)); //output: 28

console.log(addNum(5,'23')); //output: 523

6. **Can you give an example of how to use "readonly" properties in TypeScript?**

In typescript if we put ‘readonly’ before a type, we cannot change this type’s value later. Here is a example of how to use ‘readonly’ property in typescript:

let cars : {

readonly brand : string, //-- it is read only data. we cannot change it later after assigning first time.

cc : number,

sports : boolean,

country : string

} = {

brand : 'Mercedez', //-- it is read only data. we cannot change it later.

cc : 2500,

sports : false,

country : 'england'

}

console.log(cars.cc); //output: 2500

cars.cc = 3400 ; // we can change it, cz it's not read only

console.log(cars.cc); //output: 3400

cars.brand = 'audi' // we cannot change it, cz it's read only property

7. **Explain what a union type is in TypeScript and provide an example of its usage.**

A union type in TypeScript allows a variable to have multiple types. It is declared by the ‘ | ’ symbol between the types. This enables us to work with values that can be of different types. Here is an example of union type in typescript :

type License = number | string | boolean

let car : License = 'arabian license'; // we can assign it as string

car = 555 ; // we can change it to number

car = true ; // or we can change it to boolean