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'