Seção 18 TypeScript – Tipos básicos

* Sintaxe de type annotations no typescript

// Tipos básicos (aqui ocorre inferência de tipos)

let nome: string = 'Luiz'   // Qualquer tipo de strings: '' "" ``

let idade: number = 78      // 10, 1.57, -5.55, 0xf00d, 0b1010, 0o7744

let adulto: boolean = true  // true ou false

let simbolo: symbol = Symbol('qualquer-symbol') // Symbol

let big: bigint = 10n       // bigint

// Arrays

let arrayDeNumeros: Array<number> = [1, 2, 3]

let arrayDeNumeros2: number[] = [1, 2, 3]

let arrayDeString: Array<string> = ['a', 'b', 'c']

let arrayDeString2: string[] = ['a', 'b', 'c']

// Objetos

let pessoa: {nome:string, idade:number, adulto?:boolean} = {

    idade: 30,

    nome: 'Luiz'

}

// Funções

function soma(x:number, y:number):number {

    return x + y

}

const soma2: (x:number, y:number) => number = (x, y) => x + y

* Tipo void

Quando uma função ou método não retorna nada.

function semRetorno(...args:string[]){

    console.log(args.join(' + '));

}

//// Luiz + Otavio

Não retorna nada apenas está exibindo o que foi pedido

Seja explicito nos retornos de funções e métodos

* Tipo objeto (objeto em geral)

const objetoA = {

    chaveA: 'Valor A',

    chaveB: 'Valor B'

}

objetoA.chaveA = 'Outro valor'

objetoA.chaveC = 'Nova Chave' //ERRO

uma vex que o tipo foi inferido você não consegue mais alterar o tipo dele

const objetoA: {

    chaveA: string;

    chaveB: string;

    chaveC?: string; // opcional

    [key: string]: unknown; // aceita qualquer valor

} = {

    chaveA: 'Valor A',

    chaveB: 'Valor B'

}

objetoA.qualquer = 'coisa'

objetoA.outra = 1234

console.log(objetoA);

{

  chaveA: 'Valor A',

  chaveB: 'Valor B',

  qualquer: 'coisa',

  outra: 1234

}

readonly chaveA: string;

para não alterar mais o valor dessa chave

const objetoA: {

    readonly chaveA: string;

} = {

    chaveA: 'Valor A',

    chaveB: 'Valor B'

}

objetoA.chaveA = 'Outro valor' // ERRO NÃO da para alterar essa

* Tipo Array

// Array <T> - T[]

export function multiplicaArgs(...args:Array<number>):number{

    return args.reduce((ac, valor) => ac \* valor, 1)

}

export function concatenaString(...args: string[]):string{

    return args.reduce((ac, valor) => ac + valor)

}

export function toUpperCase(...args:string[]):string[]{

    return args.map((valor) => valor.toUpperCase())

}

const resut = multiplicaArgs(1, 2, 3)

const concatenacao = concatenaString('a', 'b', 'c')

const upper = toUpperCase('r', 'o', 'd')

console.log(resut);

console.log(concatenacao);

console.log(upper);

6

abc

[ 'R', 'O', 'D' ]

* Tipo tuple

Tipo exclusivo do typescript, ele não existe no javascript

// Tuple

const dadosCliente1: readonly [number, string] = [1, 'Luiz']

const dadosCliente2: [number, string, string] = [2, 'Mario', 'Solto']

const dadosCliente3: [number, string, string?] = [4, 'Luiz']

const dadosCliente4: [number, string, ...string[]] = [8, 'Carlos', 'Volto', 'JN', 'PodCast']

// dadosCliente1[0] = 100

// dadosCliente1[1] = "300"

console.log(dadosCliente1);

console.log(dadosCliente2);

console.log(dadosCliente3);

console.log(dadosCliente4);

// readonly array

const array1: readonly string[] = ['Luiz', 'Otavio']

const array2: ReadonlyArray<string> = ['Maria', 'Julia']

console.log(array1);

console.log(array2);

* Tipos null e undefined

let x;

if(typeof x === 'undefined') x = 20

// console.log(x \* 2);

export function createPerson(firstName: string, lastName?: string): {

    firstName: string;          // A saida será string

    lastName?: string;          // A saida será string

} {

    return {

        firstName,

        lastName

    }

}

export function squareOf(x: any) {

    if(typeof x === 'number') return x \* x;

    return null

}

const squareOfTwoString = squareOf(2)

if(squareOfTwoString === null) {

    console.log('Conta invalida');

} else {

    console.log(squareOfTwoString \* 100);

}

* Tipo never

Aquela função ou método numca vai retonar nada

Tipo: laço infinito ou função que lançam ERRO

export function criaErro(): never {

    throw new Error('Erro qualquer')

}

criaErro()

* Tipo Enum

No javascript não existe esse tipo, usado para quando precisamos enumera os valores

enum Cores {

    VERMELHO,

    AZUL,

    AMARELO,

}

console.log(Cores)

////{

  '0': 'VERMELHO',

  '1': 'AZUL',

  '2': 'AMARELO',

  VERMELHO: 0,

  AZUL: 1,

  AMARELO: 2

}

console.log(Cores.VERMELHO);

console.log(Cores[0]);

//// 0

//// VERMELHO

enum Cores {

    VERMELHO = 10,

    AZUL = 100,

    AMARELO = 200,

}

enum Cores {

    ROXO = 'ROXO',

    VERDE = 201,

    ROSA,// 202

}

function escolhaACor(cor:Cores):void{

    console.log(Cores[cor]);

}

escolhaACor(12345) // undefined

* Tipo unknown

O tipo unknown não deixa usar o tipo antes que ele seja verificado

let x: unknown;

x = 100

x = 'luiz'

x = 900

x = '10'

x = 10

const y = 800

if(typeof x === 'number') console.log(x + y);//checando o tipo de x

unknown pode receber qualquer valor, mais so poderar ser usado se antes essa variável for checada

* Union Types

Quando uma variável, parâmetro um retorno de função pode ter mais de um tipo

Nesse caso a e b podem ser tanto number quanto string

function addOrConcat(a: number | string, b: number | string){

    if(typeof a === 'number' && typeof b === 'number') return a + b

    return `${a}${b}`

}

console.log(addOrConcat(10, 20));

console.log(addOrConcat('10', '20'));

console.log(addOrConcat('10', 20));

console.log(addOrConcat(10, '20'));

30

1020

1020

1020

* Tipos lirerais

let x = 10  // number inteiro

x = 0b1010  // number binario

const y = 10

// let a: 100 = 100  // eslint-disable-line

// a = 120           // essa variavel só poderá ter o valor 100

let a = 100 as const // eslint-disable-line

const pessoa = {

    nome: 'Luiz' as const,

    sobrenome: 'Miranda'

}

function escolhaCor(cor: 'Vermelha' | 'Amarelo' | 'Azul'):string {

    return cor

}

console.log(escolhaCor('Vermelha'));

// Module mode

export default 1

* Types alias

Criar uma apelido para um tipo

type Idade = number

type Pessoa = {

    nome:string

    idade:Idade

    salario: number

    corPreferida?: string

}

type CorRGB = 'Vermelho' | 'Verde' | 'Azul'

type CorCMYK = 'Ciano' | 'Magenta' | 'Amarelo' | 'Preto'

type CorPreferida = CorRGB | CorCMYK

const pessoa: Pessoa = {

    idade: 30,

    nome: 'Luiz',

    salario: 200\_000

}

export function setCorPreferida(pessoa: Pessoa, cor: CorPreferida): Pessoa {

    return {...pessoa, corPreferida: cor}

}

console.log(setCorPreferida(pessoa, 'Azul'));

console.log(pessoa);

{ idade: 30, nome: 'Luiz', salario: 200000, corPreferida: 'Azul' }

{ idade: 30, nome: 'Luiz', salario: 200000 }

* Intersection types

// | - OU

// & - AND

type TemNome = { nome: string }

type TemSobrenome = { sobrenome: string }

type TemIdade = { idade: number }

type Pessoa = TemNome & TemSobrenome & TemIdade // AND

type AB = 'A' | 'B'

type AC = 'A' | 'C'

type AD = 'D' | 'A'

type Intersecao = AB & AC & AD   // A que aparece em todos

const pessoa: Pessoa = {

    nome: 'Luiz',

    sobrenome: 'Otávio',

    idade: 30

}

console.log(pessoa);

//Module Mode

export {pessoa}

{ nome: 'Luiz', sobrenome: 'Otávio', idade: 30 }

* Funções com tipo

type mapStringsCallback = (item: string) => string

export function mapStrings(array: string[], callbackfn: mapStringsCallback):string[] {

    const newArray: string[] = []

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

        const item = array[i]

        newArray.push(callbackfn(item))

    }

    return newArray

}

const abc = ['a', 'b', 'c']

const abcMapper = mapStrings(abc, item => item.toUpperCase())

console.log(abc);

console.log(abcMapper);

[ 'a', 'b', 'c' ]

[ 'A', 'B', 'C' ]