

**Мануал по TypeScript**, ноябрь 2019

Язык программирования TypeScript служит полноценной заменой JS, притом TS полноценно компилируется в JS. Он является смесью JS и C# с Java.

В нем статическая типизация и полноценное ООП.

**Объявление**

Объявить переменную можно через let или через const, создав константу.

let n1: number = 12

После имени идет тип переменной. В данном случае тип можно было отпустить и компилятор его установит.

**Типы данных**

В TS сохранились все типы из js, но он добавил еще несколько и видоизменил массивы. Массив типизирован и его можно определить двумя способами:

let arr: number[] = [12, 43, 65, 23]

let arr: Array<number> = [12, 43, 65, 23]

Новый тип данных – это *кортеж*. В нем фиксированная ширина и типы:

let tuple: [string,number,number[]] = ['Alex', 13, [43, 54, 53]];

tuple[0] = 'Kate'

tuple[2].push(13, 5)

console.log (tuple)

Для того чтоб сделать переменную динамичной можно использовать any:

let variable: any = 13

variable = 'Hello'

variable = [13]

console.log (variable)

**Пользовательские типы**

При помощи type можно создать пользовательский тип на основе других типов. К примеру id может быть строкой или числом, поэтому следует сделать тип, который может быть строкой или числом:

type ID = number | string

let myID: ID = '13243'

let friendID: ID = 13232

function handlerID(id: ID) {

console.log (id);

}

handlerID (1243353)

handlerID ('356434')

**Возврат из функции**

В ts возврат функции нужно явно указывать. Существует два специальных значения. Если функция ничего не возвращает, но притом она заканчивается, то нужно использоваться void. Если функция кидает ошибку или в ней бесконечный цикл, то нужно вернуть never.

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

return a + b

}

function log(message): void {

console.log (message)

}

function infinite(): never {

while(true) {

}

}

function throwerr(): never {

throw new Error()

}

**Интерфейсы**

*Интерфейсы* также можно использовать с объектами. Они представляют собой «объект», в котором мы указываем свойства, которые мы хотим, видит в объектах, реализующих этот интерфейс.

interface Rect {

readonly id: number,

color ?: string,

size: {

width: number,

height: number

}

}

В этом интерфейсе мы использовали readonly свойство, которое позволяет указать значение лишь один раз, похож на константу. После мы использовали необязательный параметр с помощью «?».

Пример реализации интерфейса у объекта:

let r1: Rect = {

id: 12,

color: '#fff',

size: {

width: 123,

height: 140

}

}

r1.color = '#ccc'

r1.id = 12

**Наследование интерфейсов**

Интерфейсы также могут расширяться.

interface IAnimal {

eat(food): void,

mass: number,

color: string[]

}

interface IMammalia extends IAnimal {

feedMilk(): void

}

Здесь имена интерфейсов начинаются с I и это служит для того чтоб разработчикам было более понятнее.

**Динамические интерфейсы**

Когда элементов много и они однотипны, как например стили нужно использовать *динамические интерфейсы*. Это интерфейс, где мы явно указываем «шаблон» валидных значений.

interface Styles {

[key:string]: string|number,

}

let css: Styles = {

margin: '15px',

zIndex: 1000,

opacity: 0.9,

color: '#ccc'

}

**Перечисление**

В TS можно создавать перечисление. По-умолчанию значения константам дадутся по их индексу, но после имени можно указать значение. Также у перечисления есть квадратные скобки, через, которые можно искать по значениям.

enum Season {

Winter,

Spring,

Summer,

Autumn

}

console.log (Season.Autumn)

console.log (Season [2])

enum Sociable {

Facebook = 'Facebook',

Vk = 'Vk',

Youtube = 'Youtube'

}

console.log (Sociable.Vk);

console.log (Sociable ['Vk']);

В общем-то, они не такие мощные, как в Java, что печалит.

**Перегрузка функций**

function add(a: number, b: number): number

function add(a: string, b: string): string

function add(a: any, b: any): any {

return a + b

}

console.log (add ('Hello,', 'World'))

console.log (add (12, 142))

**Синтаксис классов**

Синтаксис классов стандартный и во многом похож на классы JS 6, за исключением в том, что this привязан к классу.

class Pet {

name: string

old: number

constructor (name: string, old: number) {

this.name = name

this.old = old

Pet.count++

}

getInfo(): string {

return this.name + ': ' + this.old

}

private static count = 0

static getCount(): number {

return Pet.count;

}

}

let p1 = new Pet ('Шарик', 3)

let p2 = new Pet ('Кнопка', 10)

let p3 = new Pet ('Наташа', 30)

let str = p1.getInfo()

console.log (str)

console.log (Pet.getCount ())

**Наследование**

Наследование осуществляется через extends. Все параметры оригинального конструктора нужно инициализировать через super. Из него же можно вызывать свойства из базового класса.

class User {

name: string

isMan: boolean

constructor(name: string, isMan: boolean) {

this.name = name

this.isMan = isMan

}

getInfo(): void {

console.log(`name ${this.name};\nisMan ${this.isMan};`)

}

}

class WorkerUser extends User {

company: string

constructor(name: string, isMan: boolean, company:string) {

super(name, isMan)

this.company = company

}

getInfo() {

super.getInfo()

console.log(`company: ${this.company};`)

}

}

let alex = new WorkerUser('alex', true, 'Google')

alex.getInfo()

**Модификаторы доступа**

* Модификатор *private* можно использовать только в классе.
* Модификатор *protected* можно использовать в классе и в его подклассе
* Модификатор *public* доступен всем.

По-умолчанию свойству выдается public.

**Геттеры и сеттеры**

class Name {

private \_name: string

private \_lastname: string

public get name(): string {

return this.\_name + ' ' + this.\_lastname

}

public set name(val: string) {

let arr = val.split(' ')

this.\_name = arr[0]

this.\_lastname = arr[1]

}

}

**Абстрактные классы**

abstract class Figure {

abstract getArea(): void

}

class Rectangle extends Figure{

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

super();

}

getArea(): void{

let square = this.width \* this.height;

console.log("area =", square);

}

}

let obj = new Rectangle(20, 30)

obj.getArea();

**Реализация интерфейса**

interface IClosable {

close(): void

}

interface IAppendable {

append(value): void

}

class Data implements IClosable, IAppendable {

data: any[]

isClose: boolean = false

getByIndex(index): any {

return this.data[index]

}

close(): void {

this.isClose = true

}

append(value): void {

this.data.push (value)

}

}

**Дженерики**

Их существенно подрезали:

* вывод типов
* снижающий тип
* маски

Пример использования в классе:

class MyGen<T>{

ob: T;

constructor (ob: T) {

this.ob=ob;

}

getOb(): T{

return this.ob;

}

showType (): void {

console.log (typeof this.ob);

}

}

let gen = new MyGen<number>(12)

console.log (gen.getOb ())

gen.showType ()

Их также можно использовать в интерфейсах, абстрактных класса и методах.

**Ограничение дженериков**

interface IUser {

getInfo();

}

class User implements IUser {

\_id: number;

\_name: string;

constructor (id: number, name: string) {

this.\_id = id;

this.\_name = name;

}

getInfo() {

console.log ("id: " + this.\_id + "; name: " + this.\_name);

}

}

class Employee extends User {

\_company: string;

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

super(id, name);

this.\_company = company;

}

getInfo() {

console.log ("id: " + this.\_id + "; name: " + this.\_name+"; company:"+this.\_company);

}

}