



103,00

Рейтинг

NIX Solutions

Компания



NIX_Solutions 18 мая 2016 в 10:24

Основы синтаксиса TypeScript

Блог компании NIX Solutions, JavaScript, ООП, Программирование



В 2012 году разработчики C# из компании Microsoft создали язык [TypeScript](#) — надмножество JavaScript. Он предназначен для разработки больших приложений, от 100 тысяч строк. Давайте на примерах рассмотрим синтаксис TypeScript, его основные достоинства и недостатки, а также разберём способ взаимодействия с популярными библиотеками.

Кому это будет полезно: Web-разработчикам и разработчикам клиентских приложений, интересующимся возможностью практического применения языка TypeScript.

Существуют различные инструменты, которые позволяют писать компилируемый в JavaScript код: CoffeeScript, Dart, Uberscript и другие. К ним относится и язык программирования TypeScript, позволяющий исправить некоторые недостатки, присущие JavaScript.

Недостатки JavaScript

- Отсутствие модульности — могут возникать проблемы из-за конфликта больших файлов.
- Нелогичное поведение.
- Динамическая типизация — нет IntelliSense, из-за чего мы не видим, какие ошибки будут возникать во время написания, а видим их только во время исполнения программы.

Основные достоинства TypeScript

- Статическая типизация.
- Компилируется в хороший JavaScript.
- Наличие инструментов для разработчика.
- Нативный код хорошо читается, в нём легко разобраться.
- TypeScript поддерживается очень многими инструментами для разработки: Visual Studio, PHP Storm и другие IDE.
- Поддерживается ECMAScript 6.0.
- Компилятор найдёт и выдаст ошибку несоответствия типов до начала компилирования.

Типы переменных, которые поддерживает TypeScript

1. Number

ИНФОРМАЦИЯ

Дата основания	1994 год
Локация	Харьков Украина
Сайт	nixsolutions.com
Численность	1001–5000 человек
Дата регистрации	25 февраля 2015 г.

FACEBOOK

TWITTER

nixsolutions 19 h
120 чудесных секунд нашей первой мультиконференции!
<https://t.co/gyjTHIFY14>

nixsolutions 6 d
Совсем недавно мы объединили несколько IT-конференций в самый большой харьковский митап, и теперь с радостью пригла...
<https://t.co/LV9Xr7blOm>

nixsolutions 13 d
Разбираешься в React Native? У нас есть для тебя интересное предложение! <https://t.co/RUWe7eGzSL>
<https://t.co/9zQr2PEAxz>

[Читать @nixsolutions](#)

ВКОНТАКТЕ

БЛОГ НА ХАБРЕ

2. String
3. Boolean
4. Array
5. Enum
6. Any
7. Void

Они используются для объявления переменных, функций, классов, Generic-типов и других конструкций.

Функции

Параметры функций подразделяются на необязательные и по умолчанию.

- Необязательный параметр

```
function имя_функции (имя_переменной?:тип): тип_возвращаемого_значения
```
- Параметр по-умолчанию

```
function имя_функции (имя_переменной?:тип = "значение"):тип_возвращаемого_значения
```
- Однотипные параметры

```
function имя_функции (...имя_переменной?:тип ):тип_возвращаемого_значения
```

По аналогии с C# знак вопроса после переменной означает, что её значение можно не передавать.

Объявление функции

```
[csharp]
function getCarName(manufacturerName: string, model?: string): string {
    if (model) {
        return manufacturerName + " " + model;
    }
    return manufacturerName;
}
[/csharp]
```

Функции обратного вызова

Мы также можем передавать в качестве параметра функцию.

```
[csharp]
function numberOperation(x: number, y: number, callback: (a: number, b: number) => number) {
    return callback(x, y);
}

function addCallBackNumbers(x: number, y: number): number {
    return x + y;
}

function multiplyNumbers(x: number, y: number): number {
    return x * y;
}

console.log(numberOperation(5, 5, addCallBackNumbers)); // 10
console.log(numberOperation(5, 5, multiplyNumbers)); // 25
[/csharp]
```

Вызывая функцию `numberOperation`, мы передаём два параметра и функцию в качестве `callback`'а.

Объединение и перегрузка функций

Несмотря на строгую типизацию, в TS есть возможность использовать одну и ту же функцию с разными типами передаваемых значений. Например, в зависимости от типа переданного значения, одна функция конкатенирует наши числа, а вторая складывает.

Демистифицируем свёрточные нейросети		
7,8k	5	
Проект «Прометей»: поиск пожаров с помощью ИИ		
2,6k	6	
Нейросетевой синтез речи с помощью архитектуры Tacotron 2, или «Get alignment or die tryin'»		
2,3k	2	
Архитектуры нейросетей		
15,5k	7	
Большая конференция NIXMultiConf (Харьков)		
1,1k	0	
Конференция ThinkJava #8 в Харькове		
1,6k	0	
Делаем проект по машинному обучению на Python. Часть 3		
8,1k	1	
Делаем проект по машинному обучению на Python. Часть 2		
11,3k	4	
Делаем проект по машинному обучению на Python. Часть 1		
24,1k	4	
Создаём простую нейросеть		
52,3k	17	

```
[csharp]
//передаём 2 параметра string и получаем строку
function addOvverload(x: string, y: string): string;
//передаём 2 параметра int и получаем int результат
function addOvverload(x: number, y: number): number;

function addOvverload(x, y): any {
    return x + y;
}
[/csharp]
```

ООП. Классы

```
Class имя_класса {
    свойства;
    методы();
    constructor(свойства); }
```

```
[csharp]
class Car {
    var mazda = new Car(1, "Mazda", "6");
    console.log(mazda.getCarInfo());

    class Car {
        // объявляются три поля
        id: number;
        name: string;
        model: string;

        // инициализируется конструктор, создающий модель
        constructor(carId: number, carModel: string, model: string) {
            this.id = carId;
            this.name = carModel;
            this.model = model;
        }

        // будет отображать информацию о автомобиле
        getCarInfo(): string {
            return "Car model = " + this.name + " model= " + this.model;
        }
    }

    var mazda = new Car(1, "Mazda", "6");
    console.log(mazda.getCarInfo());
[/csharp]
```

ООП. Статические свойства и функции

Для определения статических функций и свойств используется ключевое слово `static`.

```
[csharp]
class Formula {
    static PI: number = 3.14;
    static Half = 0.5;
    // рассчитывается площадь круга
    static getCircleSquare(radius: number): number {
        return this.PI * radius * radius;
    }
    // рассчитывается площадь треугольника
    static getTriangleSquare(length: number, height: number): number {
        return this.Half * length * height;
    }
}

// созд. объект класса Formula и рассчитываются значения
var circle = Formula.getCircleSquare(16);
var triangle = Formula.getTriangleSquare(4, 7);
console.log("Площадь круга = " + circle);
console.log("Площадь треугольника = " + triangle);
[/csharp]
```

ООП. Наследование

Одним из ключевых элементов ООП является наследование, которое в TS реализуется с помощью ключевого слова `extends`. При помощи `extends` мы можем наследовать от базового класса и описать классы наследники.

```
[csharp]
interface IAnimal {
    // свойства интерфейса — два поля и один метод
    name: string;
    danger: number;
    getInfo(): void;
}

class Animal implements IAnimal {
    // наследуемся от реализованного интерфейса IAnimal
    name: string;
    danger: number;

    constructor(name: string, danger: number) {
        this.name = name;
        this.danger = danger;
    }

    getInfo(): void {
        console.log("Класс Животное. Имя: " + this.name + ", опасность: " + this.danger);
    }
}

class Fox extends Animal {
    tailLength: number;

    constructor(name: string, danger: number, tailLength: number) {
        super(name, danger);
        this.tailLength = tailLength;
        // ключевое слово super — вызывает конструктор базового класса,
        // инициализирует объект с начальными описанными в нём свойствами,
        // а потом инициализируются свойства класса наследника.
    }

    getInfo(): void {
        super.getInfo();
        console.log("Класс Лиса. Длина хвоста: " + this.tailLength + " м");
    }
}

var goose: Animal = new Animal("Гусь", 1);
goose.getInfo();

var fox: Animal = new Fox("Фоксик", 10, 1);
fox.getInfo();
[/csharp]
```

ООП. Интерфейсы

Для определения кастомного типа данных без реализации в TS (и не только) используются интерфейсы. Чтобы объявить интерфейс, используется ключевое слово `Interface`.

```
[csharp]
module InterfaceModule {
    // объявляется интерфейс IAnimal и описываются основные поля и методы этого класса
    // и потом они реализуются непосредственно на классах наследниках.
    interface IAnimal {
        name: string;
        danger: number;

        getInfo(): string;
    }

    class Animal implements IAnimal {
        name: string;
        danger: number;
    }
}
```

```

    constructor(name: string, danger: number) {
        this.name = name;
        this.danger = danger;
    }

    getInfo(): string {
        return this.name + " " + this.danger;
    }
}

var seal: IAnimal = new Animal("Тюлень", 1);
console.log(seal.getInfo());
}
[/csharp]

```

ООП. Инкапсуляция

Для сокрытия внешнего доступа к состоянию объекта и управления доступом к этому состоянию, в TS используется два модификатора: `public` и `private`.

Внутри нашего класса мы можем писать недоступные извне методы и манипулировать с их помощью.

Реализуется это как в C#:

```

[/csharp]
module Encapsulation {
    class Animal {
        private _id: string;
        name: string;
        danger: number;

        constructor(name: string, danger: number) {
            // заполнение приватного поля _id при создании метода в конструкторе.
            this._id = this.generateGuid();
            this.name = name;
            this.danger = danger;
        }

        private generateGuid(): string {
            var d = new Date().getTime();
            var uuid = 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function (c) {
                var r = (d + Math.random() * 16) % 16 | 0;
                d = Math.floor(d / 16);
                return (c == 'x' ? r : (r & 0x3 | 0x8)).toString(16);
            });
            return uuid;
        }

        public getInfo(): string {
            return "Id = " + this._id + " name = " + this.name + " danger = " + this.danger;
        }
    }

    var parrot: Animal = new Animal("Кеша", 1);
    console.log(parrot.getInfo());
}
[/csharp]

```

Таким образом мы ограничили доступ к методу `generateGuid()`. По умолчанию поля и методы имеют доступ `public`.

ООП. Generic

TypeScript позволяет создавать Generic-типы.

```
function имя_функции(имя_переменной: T): T
```

Где `T` — тип, которым типизирована функция. Также TS поддерживает типизацию интерфейсов и классов.

```
Class имя_класса
```

```
Interface имя_интерфейса
```

Где `T` — тип, которым типизирована функция.

```
[csharp]
module GenericModule {
    function getId<T>(id: T) : T {
        return id;
    }
}
// Generic класс Animal, при создании мы передаём тип generic типа и устанавливаем id
class Animal<T> {
    private _id: T;

    constructor(id: T) {
        this._id = id;
    }

    getId(): T {
        return this._id;
    }
}

var cat = new Animal<number>(16);
console.log("Cat id = " + cat.getId());

var dog = new Animal<string>("2327c575-2f7c-46c3-99f2-a267fac1db5d");
console.log("Dog id = " + dog.getId());
}
[/csharp]
```

Передаём тип, устанавливаем Id для определения типа и возвращаем нашему животному информацию. Таким образом используются Generic-типы.

Модули

Одним из недостатков JavaScript является то, что большое количество файлов может пересекаться между собой, и возникают своеобразные конфликты. В TypeScript эту проблему решают модули.

Модули — это пространство имен, внутри которого можно определить классы, перечисления, переменные, функции, другие модули. Любой набор классов, интерфейсов, функций могут быть объединены в какую-то скобку и названы определённым модулем. И потом с этим модулем можно легко и просто взаимодействовать.

Для определения модуля используется ключевое слово `module`.

```
[csharp]
import test = MyTestModule.MyTestClass;
module CarModule {
    // объявляется модуль CarModule в нём есть интерфейс ICar.
    export interface ICar {
        // ключевое слово export говорит нам о том, что данный интерфейс может быть виден извне нашего модуля. Если мы уберем слово export, данный интерфейс будет виден в скобках этого CarModule.

        id: number;
        carModel: string;
        model: string;

        getCarInfo(): string;
    }

    export class Car implements ICar {
        // создается класс Car, который имплементирует наш ICar
        id: number;
        carModel: string;
        model: string; // если мы удалим это поле, то IntelliSense предупредит о том, что не описано поле model

        constructor(carId: number, carModel: string, model: string) {
            this.id = carId;
            this.carModel = carModel;
            this.model = model;
        }

        getCarInfo(): string {
            var t = new test().getInfo();
            return "Car model = " + this.carModel + " model= " + this.model + " " + t;
        }
    }
}
```

```

    }
}

let car = new CarModule.Car(16, "BA3", "2107");
console.log(car.getCarInfo());
[/csharp]

```

Изменения в терминологии: важно, что начиная с TypeScript 1.5 изменилась номенклатура. Чтобы не было разногласий с терминологией ECMAScript 2015, «внутренние модули» стали называться «namespaces», а «внешние модули» теперь просто «modules». То есть module X { однозначно с более предпочитаемым namespace X {.

Заголовочные файлы

Для связки с какими-то глобальными переменными необходимо подключать заголовочные файлы — это один из недостатков TypeScript. Чтобы установить связь с внешними файлами скриптов JavaScript в TS, необходимо использовать декларативные или заголовочные файлы с расширением *.d.ts.

Заголовочные файлы основных библиотек уже описаны, и с ними достаточно просто и легко работать. Для этого необходимо зайти на сайт и подключить нужный набор заголовочных файлов, например, jQuery. Затем объявляются основные глобальные переменные, которые используются в jQuery, и в последующем они будут использоваться в TS-файле.

```

[/csharp]
/// <reference path="../../lib/typings/jquery.d.ts" />
class Cars {
    private cars: Array<Car> = new Array<Car>();

    load(): void {

        $.getJSON('http://localhost:53923/api/Car',
            (data) => {
                this.cars = data;
                alert('данные загружены');
            });
    }

    displayUsers(): void {
        var table = '<table class="table">';
        for (var i = 0; i < this.cars.length; i++) {

            var tableRow = '<tr>' +
                '<td>' + this.cars[i].id + '</td>' +
                '<td>' + this.cars[i].name + '</td>' +
                '<td>' + this.cars[i].model + '</td>' +
                '</tr>';
            table += tableRow;
        }
        table += '</table>';
        $('#content').html(table);
    }
}

window.onload = () => {
    var cars: Cars = new Cars();
    $("#loadBtn").click(() => { cars.load(); });
    $("#displayBtn").click(() => { cars.displayUsers(); });
};
[/csharp]

```

Недостатки TypeScript

- DefinitelyTyped — в некоторых случаях отсутствуют популярные библиотеки.
- .ts .d.ts .map — большое количество дополнительных файлов после компилирования ts-файла.
- Неявная статическая типизация. Если объявим переменную типа any, то никакой пользы от статической типизации не получим.

Преимущества TypeScript

- Строгая типизация. Огромный плюс — IntelliSense, который на этапе компиляции указывает на ошибки, и их можно исправить до этапа выполнения.
- ООП. Прекрасно поддерживаются все основные принципы ООП.
- Надмножество JavaScript'a. Достаточно скопировать код из JavaScript и вставить в TS, чтобы он заработал
- Разработка сложных решений. Благодаря поддержке модульности крупные команды разработчиков могут создавать большие приложения.

Инструменты


- Visual Studio 2015.
- WebStorm, Eclipse.
- Основные тайпинги <https://github.com/DefinitelyTyped/DefinitelyTyped>.


Заключение

TypeScript является отличным враппером, который может в значительной мере повысить производительность команды разработчиков, с сохранением совместимости с существующим кодом. TypeScript поддерживается в VisualStudio 2015 и взаимодействует с ECMAScript 6.0. Пока в JavaScript нет строгой типизации, TypeScript — это отличная альтернатива, применение которой не требует значительных затрат времени на изучение.






Теги: [синтаксис TypeScript](#)

+11
154
36,3k
32





NIX Solutions 103,00
 Компания


61,0 87,2 77
 Карма Рейтинг Подписчики

[@NIX_Solutions](#)
 Пользователь

Поделиться публикацией
 





Комментарии 32

- 
token 18 мая 2016 в 11:19 -2
- Весьма и весьма спорный список недостатков:
- Отсутствие модульности уже нет
 - Нелогичное поведение по мне так в C++ гораздо больше нелогичностей
 - Динамическая типизация — нет IntelliSense, из-за чего мы не видим, какие ошибки будут возникать во время написания, а видим их только во время исполнения программы для этого есть голова
- 
alek0585 18 мая 2016 в 11:36 -2
- Насколько мне известно, Visual Studio Code умеет делать восхитительные подсказки, которые основаны на комментариях и, внимание, на том, что возвращает метод/функция. Разве этого мало?
- 
token 18 мая 2016 в 11:39 -8

Мне сложно судить, поскольку ни Visual Studio, ни воспитательными подсказками я никогда не пользовался, как то и без всего этого живется неплохо.

Riim 18 мая 2016 в 17:38

0

Если всю жизнь ничего слаще морковки не пробовать, то она будет казаться очень сладкой)

 Ixmarduk 18 мая 2016 в 13:39

0

```
const a = {  
  name: «My name»  
};  
let i: number;  
let s: string = «name»;  
i = a[s];
```

Таким же способом можно получить доступ к приватным членам класса... Код, конечно, пахнет. IntelliSense говорит, что все ОК (если бы было `i = a[«name»]`, тогда да — подсветил бы ошибку).

P.S.: За такое и сам бы по рукам бил.

 greenwald80 18 мая 2016 в 11:52

-6

Ochen' perspektivno!

 impwx 18 мая 2016 в 12:16

+3

Список недостатков высосан из пальца. Отсутствие популярных библиотек в DefinitelyTyped никак нельзя назвать недостатком языка. Опциональная типизация — одна из главных фиш языка, позволяющая постепенно переводить существующий JS-код на TS. Про «количество генерируемых файлов» я вообще молчу. Еще очень часто среди «недостатков» всплывает тот факт, что Typescript придумал Microsoft, а они — корпорация зла! **facepalm**

Единственный реальный недостаток по сравнению с голым JS — усложняется процесс сборки, нужен дополнительный шаг для компиляции скрипта. Впрочем, чаще всего сборщик все равно используется для чего-то еще, и добавить один шаг обычно очень просто.

За полтора года использования в дизайне языка встретилось несколько багов или недоработок, которые со временем оперативно пофиксили. Был только один неприятный момент, который отказались исправить: [нельзя использовать Enum в качестве ключа для Dictionary](#).

 mr47 18 мая 2016 в 13:05

0

А что думаете насчет отсутствия полной поддержки ES6 ES2015 к примеру `Object.assign`? Не пробовали ставить `target: "es6"` и потом обрабатывать babel'ом? (Вдруг есть такой опыт)

спасибо)

bromzh 18 мая 2016 в 13:09

0

Так `Object.assign` — это не синтаксис. Это должен поддерживать движок, а не язык. Если движок не поддерживает, есть полифиллы.

 mr47 18 мая 2016 в 13:17

0

Стоп. TypeScript надмножество и поддерживает ES2015, `Object.assign` — возможность языка. Я в курсе насчет отсутствия полифиллов в составе TypeScript. А вот отсутствие в интерфейсе объекта `assign` мною было замечено. (Сейчас может уже все нормально я не знаю). Но интересно сколько раз сталкивались с таким.

bromzh 18 мая 2016 в 13:32

0

<https://github.com/Microsoft/TypeScript/blob/master/lib/lib.es6.d.ts#L4403>

Всё на месте. А отсутствие `assign` могло быть связано с тем, что IDE и редакторы часто используют свои `lib.es6.d.ts`. Например, в идеевском `es6.d.ts` нет `assign`.

 mr47 18 мая 2016 в 13:44

0

Коль автора комментария пока нет, мне интересно мнение насчет "той самой боли" typings для не популярных библиотек. Бывают достаточно большие не популяртные библиотеки, как быть ?

Можно конечно:

```
declare var lib;
```

Может вы рецепты знаете какие или статью интересную на эту тему?
ps: пару месяцев пишу на `typescript` часто беспокоят всякие мелочные проблемы.
спасибо)

 **impwx** 18 мая 2016 в 14:29

0

К сожалению, автоматической магии тут не может быть. Если нет `typing`'а для библиотеки, которую вы используете, варианта действий всего три.

Идеальный вариант — написать объявление самому и отправить PR с ним в репозиторий `DefinitelyTyped`. И самому будет удобно работать, и сообществу поможет.

Самый быстрый вариант — использовать `any` и работать с библиотекой так, как бы вы писали на чистом JS — обложиться мануалами, включить автоподсказки IDE, скрестить пальцы и молиться.

Компромисс — написать самому частичное объявление, которое покрывает не всю библиотеку, а только то, что вы из нее непосредственно используете, и ждать, когда какой-нибудь альтруист не столкнется с подобной проблемой и не выберет путь #1.

 **impwx** 18 мая 2016 в 14:51

0

Именно с этим конкретным случаем не сталкивался. В спецификации сказано, что `TS` является [надмножеством ES2015](#), так что скорее всего это баг — такие вещи можно отправить им в багтрекер и их довольно быстро чинят.

Отдельный вопрос — это существование трансформаций новых фич в более старые целевые платформы. Обычно новые возможности ES2015 внедряют сразу же, но только для целевой платформы ES6, потому что генерируемый код будет практически неизменным. А уже потом добавляют трансформацию, позволяющую использовать эту возможность в более старой платформе ES3/ES5. Например, объявления переменных `let` добавили в 1.4, а использовать их в ES5 стало возможно только в 1.5. Генераторы включили в 1.7, а поддержку в ES5 ждем не раньше релиза 2.0. На самом деле, это тоже хорошо — фичи становятся доступными быстрее, а при желании можно использовать постобработку Babel'ом.

 **fetis26** 18 мая 2016 в 13:00

+4

Простите, но что за фигню вы пишете?

Нелогичное поведение.

Аргументы будут или это просто слова?

Динамическая типизация — нет `IntelliSense`, из-за чего мы не видим, какие ошибки будут возникать во время написания, а видим их только во время исполнения программы.

`IntelliSense` это как раз наоборот сильная сторона `TypeScript`, потому что IDE более точно может выводить подсказки, на основе анализа типов, а не показывать весь список идентификаторов, которые нашла.

Функции обратного вызова

От этого вообще выпал. Вы когда с JS познакомились, вчера?

Потом еще зачем-то заголовочные файлы из Си сюда приплели

TSD уже не актуален, нужно пользоваться `typings`

 **webschik** 18 мая 2016 в 13:39

+1

Не горячитесь :)

Нелогичное поведение

Динамическая типизация — нет `IntelliSense`, из-за чего мы не видим, какие ошибки будут возникать во время написания, а видим их только во время исполнения программы.

Это ведь из списка «Недостатки JavaScript».

 **Veikedo** 18 мая 2016 в 16:06

0

Можете ответить на:

«Где хранить, как обновлять, куда класть свои, включать в Solution или нет?»

НЛО прилетело и опубликовало эту надпись здесь

gearbox 18 мая 2016 в 15:12

0

>Неявная статическая типизация. Если объявим переменную типа `any`, то никакой пользы от статической типизации не получим.

Это недостаток прокладки между монитором и клавиатурой, но никак не языка.

 **Aries_ua** 18 мая 2016 в 16:06

+1

Скажите, а в чем профит TS? Простите, не ради троллинга, действительно интересно.

Посмотрел синтаксис — очень похож на Java.

Уже лет пять-шесть пишу на JS. Особо проблем с типизацией никогда не испытывал.


Но много кто хвалит TS, а читаю статью так и не вынес для себя что либо полезное.

Seeker 18 мая 2016 в 16:12

0

>Скажите, а в чем профит TS?

Самое главное: нормальная поддержка классов.

 **a553** 18 мая 2016 в 16:32

+1

Транспиляция и типизация. Много пишу на ts с самого его появления, и сейчас я не представляю, как можно писать что-то больше 30 строк на ванильном js, без типов и классов. Это ж придётся запоминать синтаксис jQuery, Express и Angular.

bromzh 18 мая 2016 в 17:30

+2

Просто статья плохая. Посмотрите примеры из оф. документации, они нагляднее.

Условно, можно разделить профит на 2 части:

- 1) Синтаксис. Typescript — это синтаксис из самых свежих спецификаций es (классы, декораторы, деструктуризация) + немного своего сахара (enums, пространства имён). Профит очевиден: меньше кода с тем же результатом.
- 2) Собственно система типов. Это можно считать лишь некой дополнительной проверкой кода до его запуска. Потому что штуки, вроде интерфейсов или `protected`-полей существуют только на этапе компиляции ts в js.

При грамотном подходе к написанию кода профитом будут:

- а) Хорошие подсказки кода;
- б) Самодокументация кода. Очень часто достаточно просто посмотреть на сигнатуру метода в `.d.ts`-файлах, чтобы понять, чего в него нужно передавать. В случае с JS во-первых, нужно найти нужный метод или документацию для него, а это порой непросто, и во-вторых, понять, какие сигнатуры в конечном итоге приемлемы, потому что очень часто, когда разбор аргументов идёт в методе, всё становится не очень очевидно.
- в) Отлавливание багов, связанных с типами ещё до запуска. Банальный пример: опечатка в имени поля. Код на TS просто не скомпилируется и этот баг всплывёт сразу. В проекте на js это может всплыть в самый неожиданный момент.

OnYourLips 18 мая 2016 в 18:06

0

Сильная строгая типизация.

Это киллер-фича, ее и одной достаточно.

Остальное в статье из пальца высосано (как будто про ES6 автор статьи не знает и просто перепечатал с лекций по TS с edX).

Statyan 18 мая 2016 в 16:51

0

У вас список типов переменных на самом деле — список примитивных типов. И в примерах только примитивные типы используются (ну и один раз функция). А ведь вся соль дженериков и типизации проявляется, когда в качестве типа используется класс.

DreamChild 18 мая 2016 в 16:51

+2

>По аналогии с C# знак вопроса после переменной означает, что её значение можно не передавать

в C# такого нет. Там есть синтаксис `T?`, раскрывающийся в `Nullable`, и означающий, что если `T` — тип-значение,

то переменная типа T? может принимать значение null. Но это не значит, что параметр этого типа можно не передавать в метод.

 **tmn4jq** 19 мая 2016 в 10:57

0

Безотносительно плюсов и минусов JS, некоторые из которых мне тоже показались спорными, просто скажу спасибо за хорошо оформленную статью и внятные примеры.

 **Veikedo** 19 мая 2016 в 16:18

0

Это всё в лучшем виде есть в официальной документации

 **webschik** 26 мая 2016 в 10:25

0

Typescript очень нравится, но для меня главная проблема с ним это невозможность использовать ESLint. [Парсер](#) еще довольно сырой и с ним возникают проблемы. Кто как решает эту задачу? :)

bromzh 26 мая 2016 в 11:32

0

<https://www.npmjs.com/package/tslint-eslint-rules>

Strate 18 июля 2016 в 20:40

+1

С Typescript развитие и эволюция кода (рефакторинг) становятся очень простыми и лёгкими — изменил сигнатуру функции, и исправляешь все использования просто глядя на диагностический вывод компилятора. Очень часто бывает такое — пишешь какую-то сложную конструкцию, TS ругается. Пытаешься разбираться и понимаешь, что если бы он не ругнулся, то эту же ошибку ты бы уже получил в рантайме, причём в виде какого-нибудь "Cannot get property of null" или что-то такое.

Короче с TypeScript почти (до стабилизации strictNullChecks) работает уравнение Компилируется === Работает.

Только [полноправные пользователи](#) могут оставлять комментарии. [Войдите](#), пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

Гаджеты с барахолки: зачем покупать 20-летний ноутбук Packard Bell за 10 евро

+58

52,4k

42

126

Для чего и как мы скрываем госномера автомобилей в объявлениях Авито

+62

33,5k

86

116

Принципы построения REST JSON API

+30

24,5k

451

187

Интервью с Владимиром Лихачевым, отцом Николая Лихачева, более известного как Крис Касперски

+67

19,4k

56

81

Автоматизация бизнес-процессов в Excel или как спасти девушку от переработок

+37

18,7k

53

42

Аккаунт

Разделы

Информация

Услуги

Приложения

[Войти](#)

[Публикации](#)

[Правила](#)

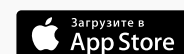
[Реклама](#)

[Регистрация](#)

[Новости](#)

[Помощь](#)

[Тарифы](#)



[Хабы](#)[Документация](#)[Контент](#)[Компании](#)[Соглашение](#)[Семинары](#)[Пользователи](#)[Конфиденциальность](#)[Песочница](#)

© 2006 – 2019 «ТМ»

[Настройка языка](#)[О сайте](#)[Служба поддержки](#)[Мобильная версия](#)