

**Вахитов Булат  
Рафисович**

# **Про TypeScript, и про нас**

Telegram - @fclm\_man

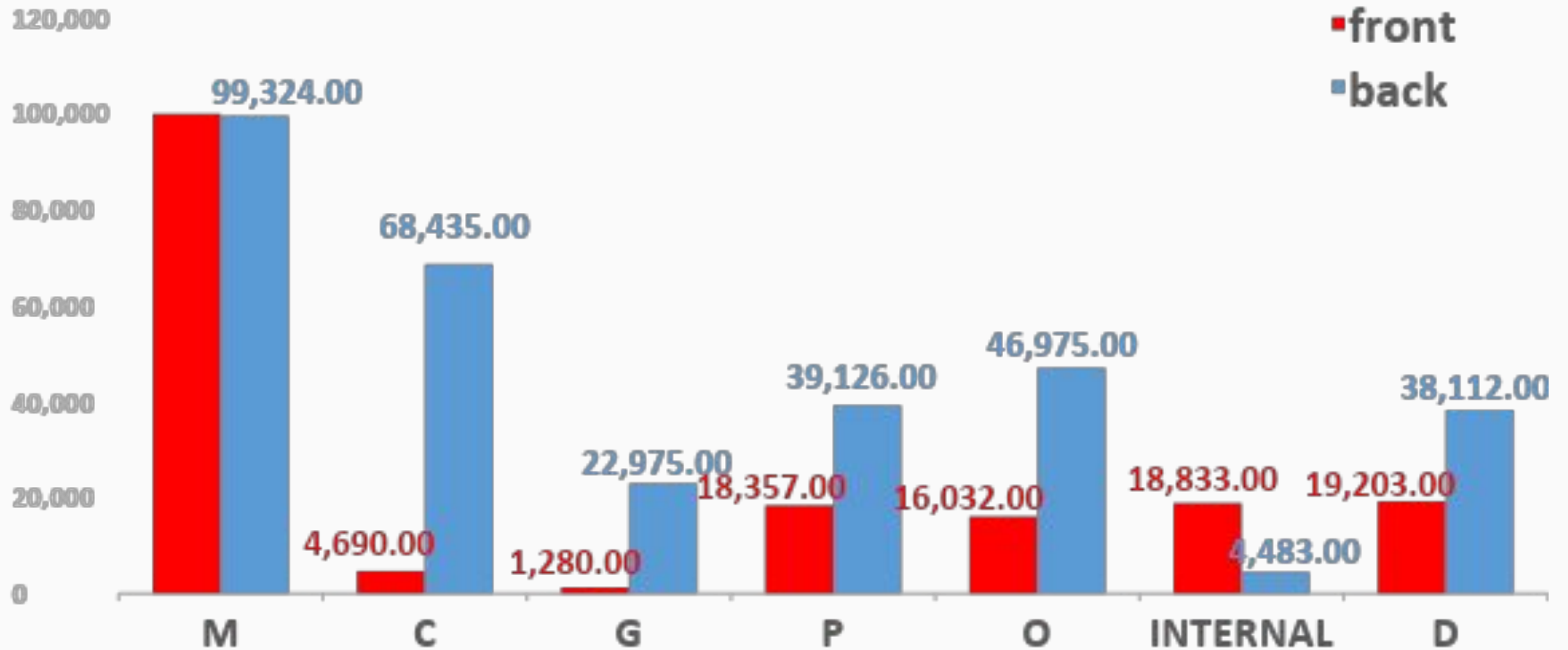
[vakhitov.br@gazprom-neft.ru](mailto:vakhitov.br@gazprom-neft.ru)

**ООО «ИТСК»**

# КТО МЫ

- ИТСК
- Системный интегратор
- Дочернее предприятие ПАО «Газпром
- ~~Пишем~~ софт, много софта

# Наши проекты (количество строк кода)



# Что это такое

- Объектно-ориентированный язык
- Представлен Microsoft в 2012 году
- Является расширением JavaScript
- Добавляет типизацию
- Добавляет модификаторы доступа членам
- Добавляет синтаксические конструкции
- Компилируется в JavaScript

Кому может пригодиться?



# Сфера применения

- Везде где можно исполнять javascript код

# Сфера применения

- При разработке приложений для браузеров

# Сфера применения

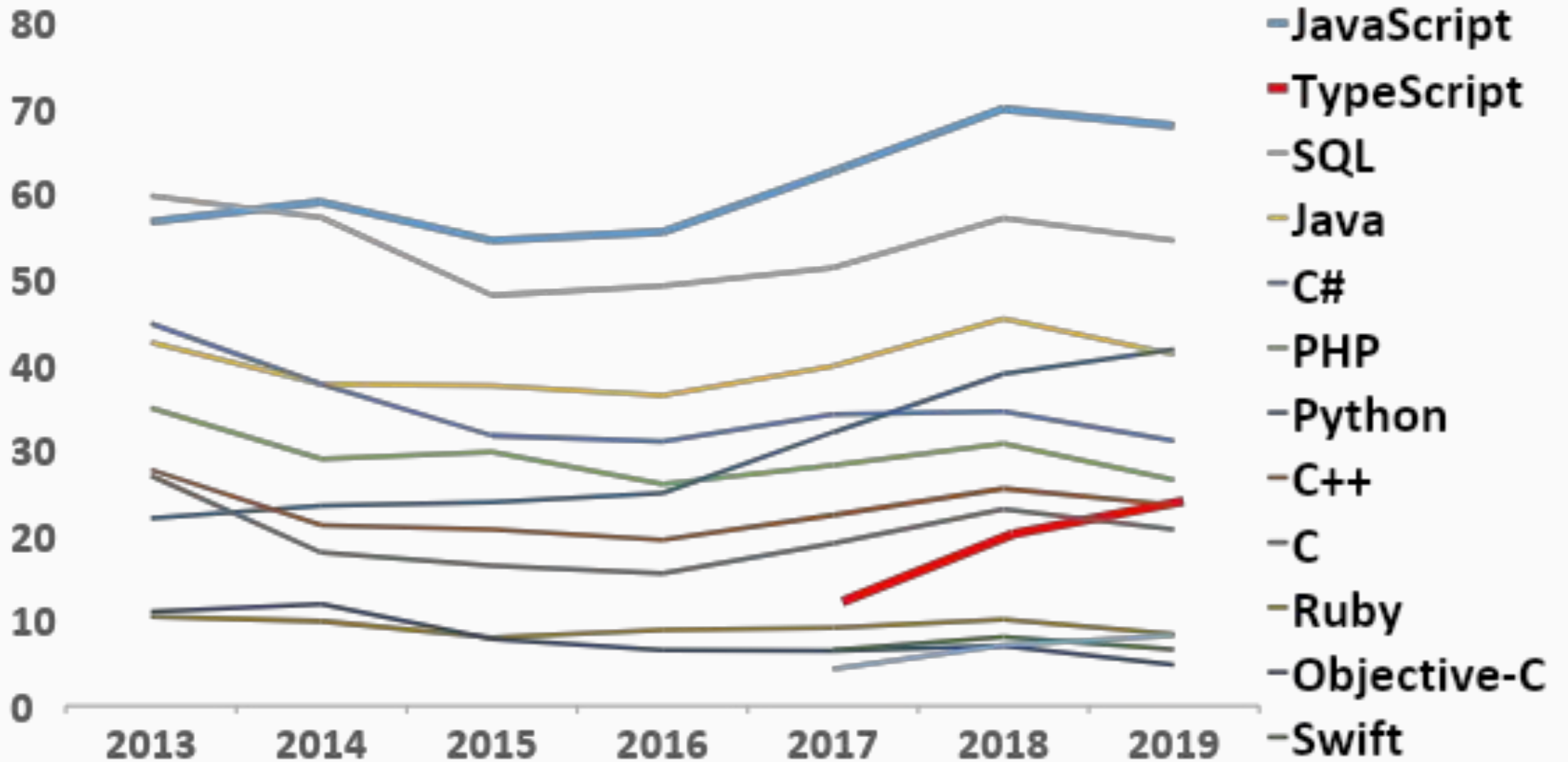
- Если вы ~~наркоман~~ fullstack developer и пишете бэк на ноде, то код скомпилированный из TS может исполняться в node.js



# Сфера применения

- Deno – новая технология от создателя node, обещает исполнять TypeScript

# Динамика популярности языков со времени релиза typescript (по мнению stackoverflow)



# Знакомство

- Наше знакомство с typescript – 2016 год
- Причины:
  - ✓ Строгая типизация
  - ✓ Выразительная система типов
  - ✓ Framework который мы выбрали для enterprise разработки пропагандирует ts

**Что делают разработчики,  
когда встречают что-то  
незнакомое?**

**Сопротивляются!**

# Стадии принятия

1. Отрицание – да кому это только пришло в голову, типы в JS???

# Стадии принятия

2. Гнев – \*»№%::\* эту строгую типизацию, это сколько надо dto наваять

# Стадии принятия

3. Торг – ок, типы. Any – тоже вполне себе тип

# Стадии принятия

4. Депрессия – ее мы, кажется, пропустили



# Стадии принятия

5. Принятие – осознание плюсов типизации, кропотливое прорабатывание типов, избавление от any

# Что он нам дает

- Привычно выглядящие ООП конструкции
- Статическую типизацию
- Классы
- Интерфейсы
- Обобщенные типы
- Типы перечисления
- Модификаторы доступа
- Декораторы

# Система типов TypeScript

- Статическая

```
1 let num: number;  
2 num = 'string';
```

# Система типов TypeScript

---

- Сильная (строгая

```
1 let x = 5;  
2 let y = '5';  
3 let result = y * x;
```

# Система типов TypeScript

- Структурная

# Номинативная ТИПИЗАЦИЯ

C#

```
1 class PointA {
2     public int X;
3     public int Y;
4 }
5
6 class PointB {
7     public int X;
8     public int Y;
9 }
10
11 class Test {
12     public int GetX(PointA point) {
13         return point.X;
14     }
15
16     /**/
17
18     var test = new Test();
19     test.GetX(new PointB());
```

# Структурная типизация

```
1 class PointA {  
2   x: number;  
3   y: number;  
4 }  
5  
6 class PointB {  
7   x: number;  
8   y: number;  
9 }  
10  
11 let getX = (point: PointA) => point.x;  
12  
13 getX(new PointB());  
14  
15 getX({  
16   x: 5,  
17   y: 7  
18 });
```

# Откуда берутся ТИПЫ В TypeScript

---

```
1 let obj = {  
2   x: 1,  
3   y: 'qwerty'  
4 };  
5  
6 obj.z = 100;  
7 // ERROR: Property 'z' does not exist  
8 // on type '{ x: number, y: string }'
```

---



# Откуда берутся ТИПЫ в TypeScript

---

```
1 let obj;  
2  
3 obj = {  
4   x: 1,  
5   y: 'qwerty'  
6 };  
7  
8 obj.z = 100; // OK
```

---

# Откуда берутся ТИПЫ в TypeScript

```
1 "compilerOptions": {  
2   "noImplicitAny" : true  
3 }
```

# Откуда берутся ТИПЫ в TypeScript

```
1 let obj: {  
2   x: string,  
3   y: number  
4 };
```

# Откуда берутся ТИПЫ В TypeScript

---

```
1 type Point = {  
2   x: string,  
3   y: number  
4 }
```

---

# Откуда берутся ТИПЫ в TypeScript

---

```
1 interface IPoint {  
2   x: string;  
3   y: number;  
4 }
```

---

# Откуда берутся ТИПЫ В TypeScript

```
1 class Point {  
2   x: string;  
3   y: number;  
4 }
```

# Куда деваются ТИПЫ В TypeScript

```
1 let x1 = {x: 'ABC'};
2
3 let x2: { x: string };
4
5 type SomeType = { x: string }
6
7 interface ISomeType {
8   x: string
9 }
10
11 class SomeClass {
12   x: string
13 }
```

# Куда деваются ТИПЫ В TypeScript

ES5

---

```
1 var x1 = { x: 'ABC' };
2 var x2;
3 var SomeClass = (function () {
4     function SomeClass() {
5     }
6     return SomeClass;
7 }());
```



# Куда деваются ТИПЫ В TypeScript

ES5

---

```
1 let x1 = { x: 'ABC' };  
2 let x2;  
3 class SomeClass {  
4 }
```

---

# Анонимный тип



Compile-time тип

# Именованный тип, Интерфейс



Compile-time тип

Compile-time ID

# Класс Typescript



Compile-time тип

Функция-  
конструктор, или es6  
класс

Compile-time ID

# Рассмотрим такой вариант

```
1 interface ISomeClient<T> {
2   get<T>(path: string): Promise<T> | T;
3 }
4
5 class SomeClient<T> implements ISomeClient<T> {
6   get<T>(path: string): Promise<T> | T {
7     // код стороннего сервиса, например HttpClient из angular
8   }
9 }
10
11 class UserProfile {
12   name: string;
13   surname: string;
14   secondName: string;
15
16   getFullName() {
17     return `${this.name} ${this.surname} ${this.secondName}`;
18   }
19
20   getSurnameLength() {
21     return this.surname.length;
22   }
23 }
```

# Ответ сервера

---

```
1 [{
2   "name": "Aleksander",
3   "patronymic": "Sergeevich"
4 }, {
5   "name": "Feudor",
6   "patronymic": "Michaylovich"
7 }]
```

# И ВОТ ЧТО МЫ ПОЛУЧИМ

---

```
1 const someClient = new SomeClient<UserProfile[]>();
2
3 let response = someClient.get<UserProfile[]>('users');
4
5 if (response && response.length) {
6     console.log(response[0].getFullName());
7     // Ivan undefined undefined
8
9     console.log(response[0].getSurnameLength());
10    // Error: cannot get property length of undefined
11 }
```

# Результирую щий js

```
1 var SomeClient = /** @class */ (function () {
2   function SomeClient() {
3   }
4
5   SomeClient.prototype.get = function (path) {
6     // Здесь код стороннего сервиса,
7     // например HttpClient из angular
8   };
9   return SomeClient;
10 }());
11 var UserProfile = /** @class */ (function () {
12   function UserProfile() {
13   }
14
15   UserProfile.prototype.getFullName = function () {
16     return `${this.name} ${this.surname} ${this.secondName}`;
17   };
18   UserProfile.prototype.getSurnameLength = function () {
19     return this.surname.length;
20   };
21   return UserProfile;
22 }());
```



## Еще про интерфейсы

```
1 interface IAmInterface {  
2   name: string;  
3   id: number;  
4 }  
5  
6 let a = {name: 'test'}  
7 if (a instanceof IAmInterface) {  
8   console.log('No way')  
9 }
```

# Еще про интерфейсы

---

```
1 10:18 - error TS2693: 'IAmInterface' only refers to a type,  
2         but is being used as a value here.  
3  
4 10 if (a instanceof IAmInterface) {  
5     ~~~~~
```

# Еще немного про интерфейсы

**A**

```
1 interface IAmInterface {
2     name: string;
3     id: number;
4 }
5
6 interface IAmInterface {
7     oneMore: Date;
8 }
```

**B**

```
1 interface IAmInterface {
2     name: string;
3     id: number;
4 }
5
6 interface IAmInterface {
7     id: string;
8 }
```

# Еще немного про интерфейсы

---

```
1 interface User {
2   name: string;
3   id: number;
4   getName(): string;
5 }
6
7 class User {
8 }
9
10 let user = new User();
11 user.getName();
```

# Модификаторы доступа, все что о них нужно знать

```
1 class IAmClass {
2     private propPrivate: string = 'pPrivate';
3     protected propProtected: string = 'pProtected';
4     public propPublic: string = 'pPublic';
5
6     private methodPrivate() {
7         return 'mPrivate';
8     }
9
10    protected methodProtected() {
11        return 'mProtected';
12    }
13
14    methodPublic() {
15        return 'mPublic';
16    }
17 }
```

# Модификаторы доступа, все что о них нужно знать

```
1 var IAmClass = /** @class */ (function () {
2   function IAmClass() {
3     this.propPrivate = 'pPrivate';
4     this.propProtected = 'pProtected';
5     this.propPublic = 'pPublic';
6   }
7   IAmClass.prototype.methodPrivate = function () {
8     return 'mPrivate';
9   };
10  IAmClass.prototype.methodProtected = function () {
11    return 'mProtected';
12  };
13  IAmClass.prototype.methodPublic = function () {
14    return 'mPublic';
15  };
16  return IAmClass;
17 }());
```

# Более того, даже в ts

```
1 class IAmClass {
2   private propertyPrivate: string = 'propertyPrivate';
3   protected propertyProtected: string = 'propertyProtected';
4   propertyPublic: string = 'propertyPublic';
5
6   private methodPrivate() {
7     console.log('private');
8   }
9
10  protected methodProtected() {
11    console.log('protected');
12  }
13
14  methodPublic() {
15    console.log('public');
16  }
17 }
18
19 let test = new IAmClass();
20 test['methodPrivate']();
```

# Generic

---

```
class GenericClass<T> {  
    property: T;  
  
    constructor(value: T) {  
        this.property = value;  
    }  
}
```



# Generic

```
class Deserializer<T> {  
    deserialize<T>(json: string): T {  
        let t = new T();  
        // десериализуем JSON  
        return t;  
    }  
}
```

# Generic

```
public interface IDeserializer<T> {  
    T Deserialize(string json);  
}
```

```
public class Deserializer<T>: IDeserializer <T> where T : new () {  
    public T Deserialize(string json)  
    {  
        var obj = new T(); //Десериализуем JSON  
        return obj;  
    }  
}
```

# C#

# Generic

---

```
class Deserializer<T> {  
    deserialize<T>(json: string, ctor: new () => T): T {  
        let t = new ctor();    // десериализуем данные  
        return t;  
    }  
}
```

```
let d = new Deserializer<SomeClass>(json, SomeClass);
```

---

# Generic

```
1 class DataRow<T> {  
2   data: T;  
3 }  
4  
5 class Column<T> {  
6   width: number;  
7   dataIndex: keyof T;  
8 }  
9  
10 class Table<T> {  
11   data: DataRow<T>[];  
12   columns: Column<T>[];  
13 }
```

# Generic

---

```
1 class User {
2   id: number;
3   name: string;
4   mail: string;
5   address: string;
6 }
7
8 const userColumn = new Column<User>();
9
10 userColumn.dataIndex = "address"; // OK
11 userColumn.dataIndex = "salary"; // ERROR
```

# Контекст

---

```
function noThis(this: void) {  
    return this.name; // Такой код не скомпилируется  
}
```

# Контекст

```
interface Contract {  
  id: number;  
  name: string;  
  surname: string;  
}
```

```
function safeFunc(this: Contract) {  
  return this.name;  
}
```

```
let wrongObject = {  
  name: 'name',  
  safeFunc: safeFunc  
};
```

```
wrongObject.safeFunc(); // Здесь ошибка
```

```
let rightObject = {  
  id: 0,  
  name: 'name',  
  surname: 'surname',  
  safeFunc: safeFunc  
};
```

```
rightObject.safeFunc(); // OK
```

# Декораторы

Для использования нужно включить `--experimentalDecorators`

---

```
declare type ClassDecorator = <TFunction extends Function>(target: TFunction) => TFunction | void;

declare type PropertyDecorator = (target: Object, propertyKey: string | symbol) => void;

declare type MethodDecorator = <T>(target: Object, propertyKey: string | symbol, descriptor:
TypedPropertyDescriptor<T>) => TypedPropertyDescriptor<T> | void;

declare type ParameterDecorator = (target: Object, propertyKey: string | symbol, parameterIndex:
number) => void;
```



# Декораторы

```
1 class DiInstance {
2   key: string;
3   value: any;
4   constructor(key: string, value: any) {
5     this.key = key;
6     this.value = value;
7   }
8 }
9
10 interface IDbConnection {
11   connectionString: string;
12   open(): boolean;
13   close(): boolean;
14 }
15
16 class DbConnection implements IDbConnection {
17   connectionString: string;
18   open() {
19     console.log(`connected to ${this.connectionString}`);
20     return true;
21   }
22   close() {
23     console.log(`disconnected акщъ ${this.connectionString}`);
24     return true;
25   }
26   constructor(connectionString: string) {
27     this.connectionString = connectionString
28   }
29 }
```

# Декораторы

---

```
1 let dbConnection = new DbConnection('db_instance');  
2  
3 const diList: DiInstance[] = [];  
4  
5 diList.push(new DiInstance('DB', dbConnection));
```

# Декоратор

```
1 function Inject(injectionToken: string) {
2   return function (target: any, key: string) {
3     const value = diList.find(_ => _.key === injectionToken);
4     if (!value) {
5       throw new Error('injectionToken not found in container');
6     }
7     Object.defineProperty(target, key, {
8       value: value.value,
9       enumerable: true,
10      configurable: true
11    });
12  }
13 }
14
15 class DiClass {
16   @Inject('DB') connection: IDbConnection;
17 }
18 let instance = new DiClass();
19
20 instance.connection.open();
```

# Выводы



- Хорошо подойдет людям с опытом backend разработки на строго типизированных языках

# Выводы



- Расширяет возможности JavaScript, добавляя фишки, которые еще не вошли в стандарт, но ожидаются в будущем

# Выводы



- Добавляет статическую типизацию

# Выводы



- Более удобная навигация по проекту и более точные подсказки IDE.

# Выводы



- Описание предметной области с помощью типов в крупных проектах позволяет лучше понимать чужой код



# Выводы



- Обещает корректность типов в вашем приложении

# Выводы



- Он выглядит как C# или Java, но ими не является, вводя в заблуждение

# Выводы

---

- Это все еще JavaScript с прототипным наследованием
- Из этого следует, что TS опасен для людей, которые плохо знают JavaScript

# Выводы



- Большая часть конструкций исчезает после компиляции

# Выводы

---

- Все что вошло в программу из нетипизированной среды может вам все поломать (json с бэка, например)

# Выводы



- Сторонние библиотеки с коллбэками, которые применяют свой контекст могут нам все поломать

# Выводы



- Проверка типов полностью отсутствует в рантайме

# Выводы



- Добавляется обязательный этап сборки проекта



# Выводы



- Не гарантирует корректность программы)

# И напоследок (с) habr

Вот чем динамическая типизация хуже статической?

Тем, что статическая типизация позволяет узнать, что ваша программа не делает ерунду до запуска вашей программы, а динамическая – нет. (с некоторыми оговорками)