Typescript!

Зачем?

Московский клуб программистов 2020

Стариченко Никита

LinkedIn: www.linkedin.com/in/nikita-starichenko/
Telegram: @nikita_starichenko

6+ Years of Experience:

STO solutions, San Francisco USA **01/2020 - now**

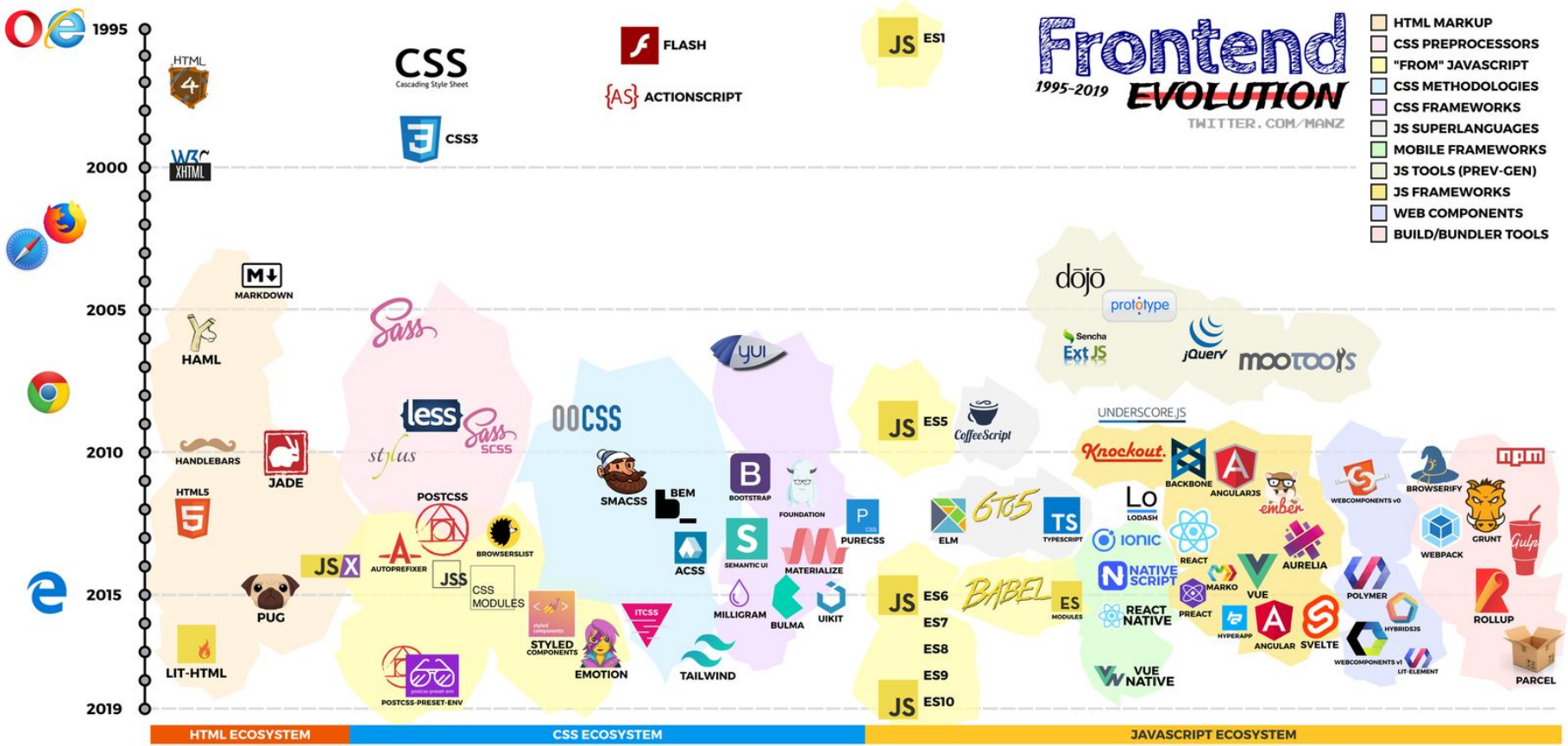
- Designing and Developing SPA application for healthcare risk adjustment automation
- Designed architecture of the entire system from scratch

Dodo Pizza, Oxford USA / Moscow Russia **03/2018 – 04/2019**

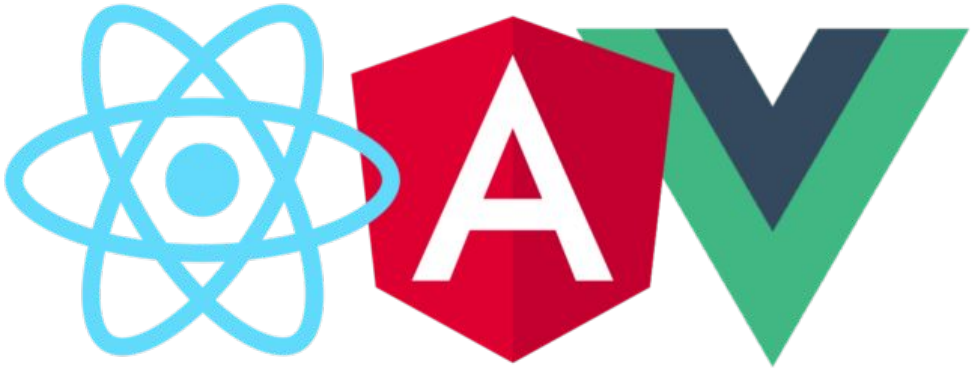
- Developing and maintaining DODO IS
- Reduced release time by 40% by fixing more than 50 UI tests that led to a decrease of manual testing
- Piloted first microservice on .Net Core and GRPC that is composing by Docker that uses a full CI/CD including integration tests and run in Kubernetes
- Piloted integration React to Angular.js and add ability to step by step rewrite frontend from Angular to React that speeded up front development by 2 times
- Got rid of the need to restart the system by eliminating the daily memory leak of 100mb by finding that leak in .Net Core application using memory snapshot tools on Linux in runtime

Vanilla JS

Modern Frontend



Modern JS



А что плохого?

1. Ошибки в данных
2. Ошибки в запросах
3. Поддерживаемость в больших проектах

Чуть нагляднее

```
> '5' - 3
< 2
```

```
> '5' + 3
< "53"
```

```
> '5' - '3'
< 2
```

```
> '5' + -'4'
< '5-4'
```

```
> var x=3
< undefined
> '5' -x + x
< 5
> '5' + x - x
< 50
```

```
> typeof NaN
< "number"
> NaN == NaN
< false
```

```
> Array(3)==",,"
< true
```

```
> typeof null
< "object"
```

```
> null instanceof Object
< false
```

```
> true + true === 2
< true
> true - true === 0
< true
```

```
> true === 1
< false
```

```
> true && NaN
< NaN
```

```
> false && NaN
< false
```

Typescript

Добавим немножко типов

Пример

Какие поля могут быть в этом объекте?

```
function getProductName(product) {
    // product.name ?
    // product.displayName ?
    // product.printName ?
}
```

Теперь уже с типами:

```
// может использоваться разными функциями
interface Product {
    displayName: string
}

function getProductName(product: Product): string {
    return product.displayName
}
```

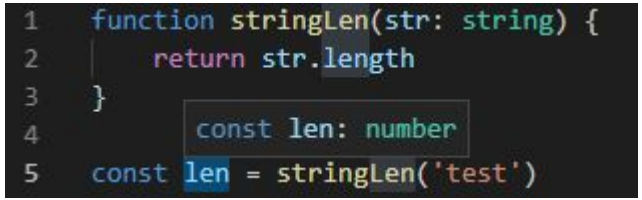
Система типов TypeScript

Есть сходства с C#. Есть интерфейсы. Есть классы с публичными и приватными методами и полями.

```typescript
interface Costable {
    getPrice(): number
}

class Pizza implements Costable {
    private name: string
    private price: number

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

    public getPrice() {
        return this.price
    }
}

const pizza = new Pizza("Пепперони", 500)
console.log(pizza.getPrice()) // 500
```

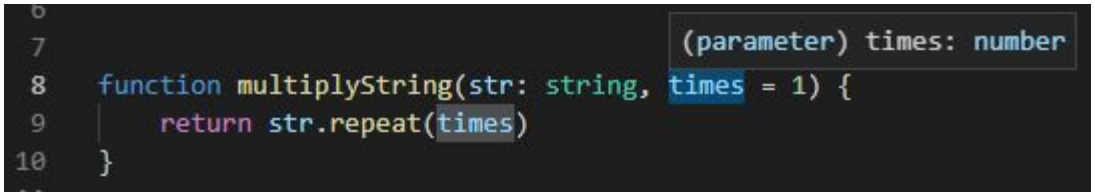
Система типов TypeScript. Вывод типов

В результатах работы функции:



Система типов TypeScript. Вывод типов

В параметрах:



Система типов TypeScript. Вывод типов

Неявно создаются типы для всех объектов:

Система типов TypeScript

Типы можно определять разными способами:

```
class Product {
    name: string
    constructor(name: string) {
        this.name = name
    }
}
const cola: Product = { name: 'Кола' }
```

```
interface Product {
    name: string
}
const cola: Product = { name: 'Кола' }
```

```
type Product = {
    name: string
}
const cola: Product = { name: 'Кола' }
```

Результат вроде одинаковый. В чем разница и как правильно?

Система типов TypeScript

Interface

```ts
interface Product {
    name: string
}
const cola: Product = { name: 'Кола' }
```

Несколько определений интерфейса сливаются

```ts
interface Product {
    name: string
}
interface Product {
    price: number
}
const cola: Product = { name: 'Кола', price: 500 }  // оба поля обязательны
```

```cs
interface IProduct
{
    0 references
    string Name { get; set; }
}
```

C#

Система типов TypeScript

Class

```typescript
interface Product {
    name: string
}
class Pizza implements Product {
    name: string
    constructor(name: string) {
        this.name = name
    }
}
const pepperoni = new Pizza('Пепперони')
console.log(pepperoni instanceof Pizza)     // true

const margarita = { name: 'Маргарита' }
console.log(margarita instanceof Pizza)     // false
```

```csharp
1 reference
interface IProduct
{
    2 references
    string Name { get; set; }
}
1 reference
class Pizza : IProduct
{
    2 references
    public string Name { get; set; }
    0 references
    public Pizza(string name)
    {
        Name = name;
    }
}
```

C#

Система типов TypeScript

Type

```typescript
type Product = {
    name: string
}
type WithPrice = {
    price: number
}
type Pizza = Product & WithPrice
const cola: Pizza = { name: 'Кола', price: 500 }
```

```fsharp
type A = int * int
type B = {FirstName:string; LastName:string}
type C = Circle of int | Rectangle of int * int
type D = Day | Month | Year
type E<'a> = Choice1 of 'a | Choice2 of 'a * 'a

type MyClass(initX:int) =
   let x = initX
   member this.Method() = printf "x=%i" x
```

F#

Interface

- описывает структуру объекта
- не существует в рантайме
- может расширять другой интерфейс или даже класс
- несколько объявлений мержатся
- имя видно на этапе компиляции

Class

- описывает структуру объекта
- существуют и в рантайме и на этапе компиляции
- может реализовывать интерфейсы
- может быть унаследован от другого класса

Type

- просто синоним для другого типа
- может быть вычисляемым
- не существует в рантайме
- не может наследовать или быть унаследованным

Но создавать свой тип под каждую модель очень долго, можно побыстрее?

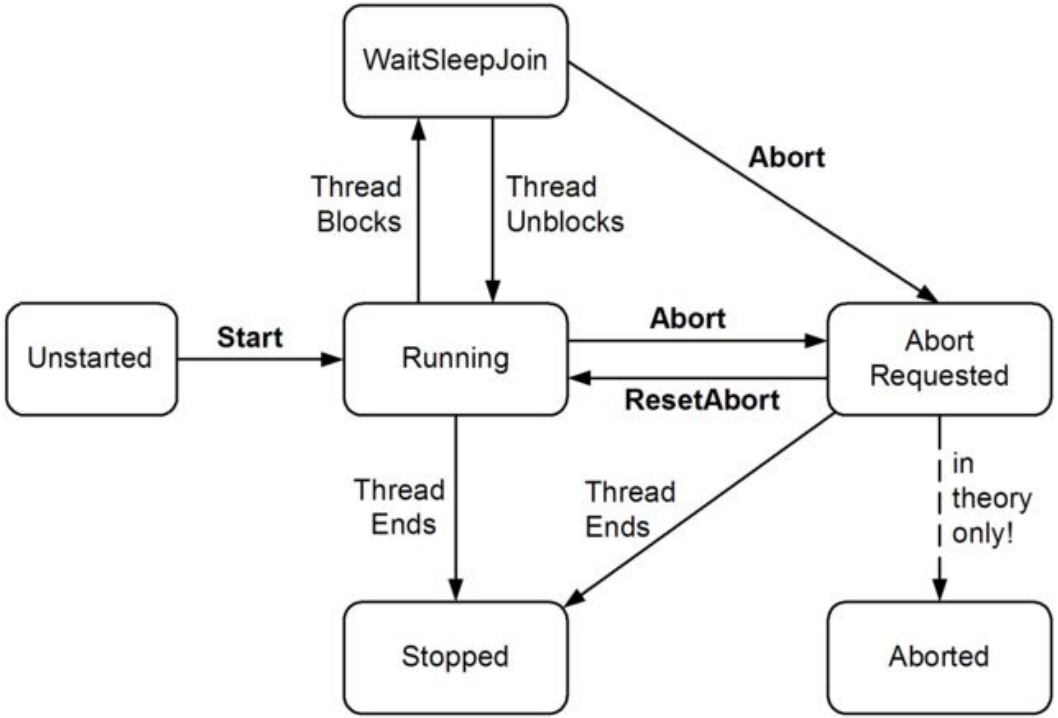
Да, вот так:

1. Пересечения
2. Объединения
3. Generics

Алгебраическая система типов

```
var test = {}
test.state = "State1"
test.state = "State2"
test.anotherState = "AnotherState"
```

JS



C#

Система типов TypeScript. Пересечения типов

Типы могут быть составными

Из нескольких типов можно собирать один, включающий в себя все поля обоих:

```ts
type WithPrice = {
    price: number,
}
type WithName = {
    name: string,
}
type WithPriceAndName = WithPrice & WithName

const item: WithPriceAndName = {
    name: "pizza",
    price: 500,
}
```

Система типов TypeScript. Объединение типов

Для случаев, когда переменная может принимать разные типы:

```
14  type StringOrNumber = string | number
15  let some: StringOrNumber
16  some = 'str'
17  some = 123
18  some = true
```

Система типов TypeScript. Generics

По сути - это параметры типов. Как и в C# можно создавать открытые обобщенные типы, а специализировать в любой момент.

```
function sort<T>(arr: T[]) {
    let len = arr.length;
    for (let i = 0; i < len; i++) {
        for (let j = 0; j < len; j++) {
            if (arr[j] > arr[j + 1]) {
                let tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
    return arr;
}

const arr1: string[] = ["Test3", "Test1", "Test2"]
const arr2: number[] = [3, 1, 2]

const sorted1 = sort<string>(arr1)
const sorted2 = sort<number>(arr2)
```

Система типов TypeScript. Generics

Помимо генерализации, можно делать вычисления. Generic типы можно представлять себе как функции с параметрами.

Например, **Identity** тип (который возвращает тип параметра, не меняя его):

```
type Identity<T> = T

type Result = Identity<number>

const r: Result = 123
```

Система типов TypeScript. Generics

Помимо генерализации, можно делать вычисления. Generic типы можно представлять себе как функции с параметрами.

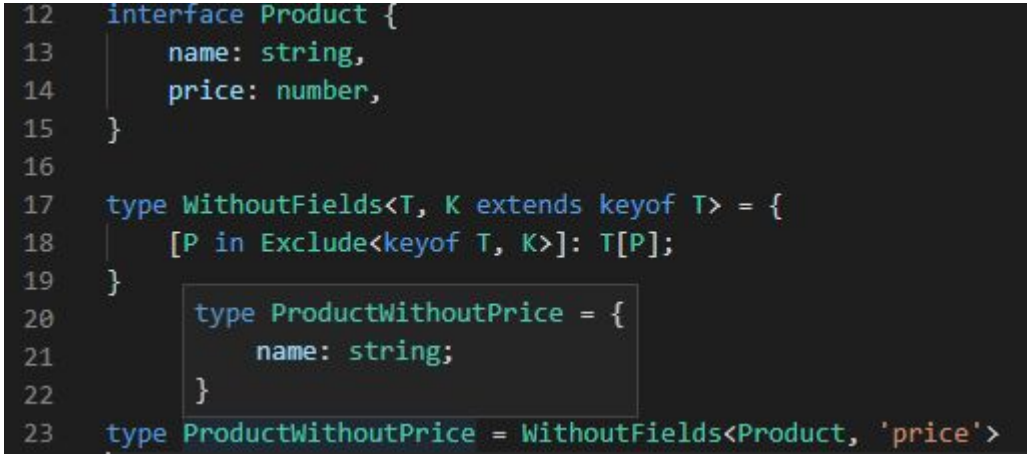
Простой пример с if:

```
type NumberOrNot<T> = T extends number ? true : false

type DefinitelyFalse = NumberOrNot<string>

const notvalid: DefinitelyFalse = true
const valid: DefinitelyFalse = false
```

DefinitelyFalse - это не конкретное значение, это тоже тип, который может принимать только одно константное значение (в данном случае **false**)

Система типов TypeScript. Generics

Пример посложнее:



Система типов TypeScript. Generics

Часто используемые:

- **Partial** - помечает все поля типа как необязательные
- **Readonly** - помечает все поля как только для чтения
- **Record** - возвращает массив ключ-значение, как Dictionary

Partial - помечает все поля типа как необязательные

```
interface Todo {
    title: string;
    description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
    return { ...todo, ...fieldsToUpdate };
}

const todo1 = {
    title: "organize desk",
    description: "clear clutter",
};

const todo2 = updateTodo(todo1, {
    description: "throw out trash",
});
```

Readonly - помечает все поля как только для чтения

Record - возвращает массив ключ-значение, как Dictionary

```ts
interface PageInfo {
    title: string;
}

type Page = "home" | "about" | "contact";

const nav: Record<Page, PageInfo> = {
    about: { title: "about" },
    contact: { title: "contact" },
    home: { title: "home" },
};
```

Система типов TypeScript. Generics

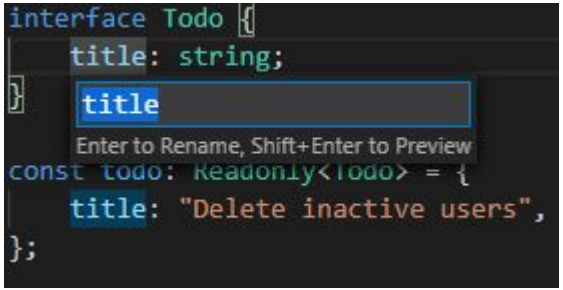
Другие часто используемые. Чтобы не копипастить из проекта в проект.

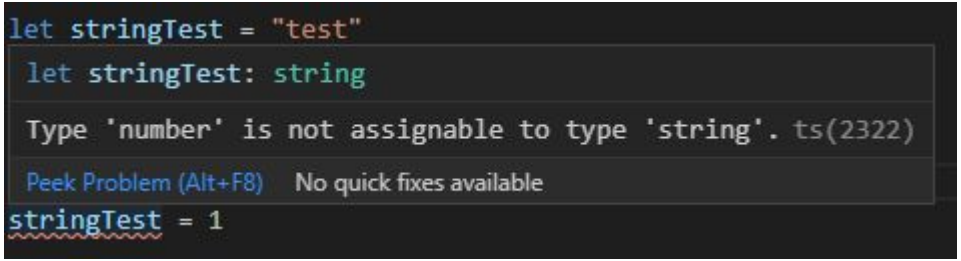
https://github.com/sindresorhus/type-fest

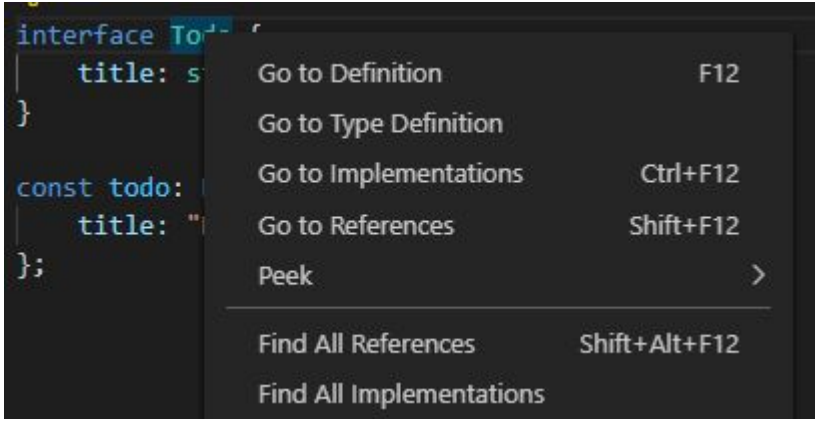
- `Omit` - Create a type from an object type without certain keys.
- `Merge` - Merge two types into a new type. Keys of the second type overrides keys of the first type.
- `MergeExclusive` - Create a type that has mutually exclusive properties.
- `RequireAtLeastOne` - Create a type that requires at least one of the given properties.
- `LiteralUnion` - Allows creating a union type by combining primitive types and literal types without sacrificing auto-completion in IDEs for the literal type part of the union. Workaround for Microsoft/TypeScript#29729.

Ну и что, стало лучше? Да.

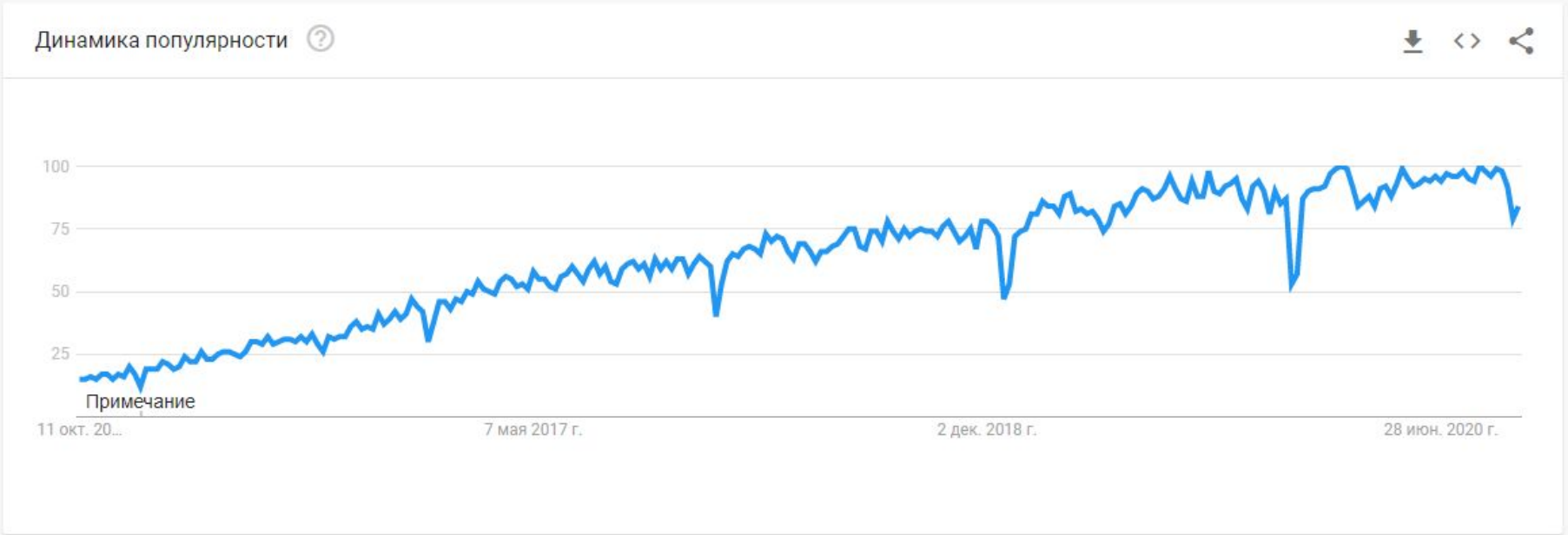
1. Намного больше ошибок выявляется на ранней стадии
2. Поддержка IDE упрощает навигацию по коду и поддержку
3. Рефакторинг намного проще







Популярность Typescript



6000+ поддерживаемых библиотек

Что использовать?

Javascript:

1. Легче изучить
2. Меньше кода
3. Быстрая разработка на старте
4. Хорош для маленьких проектов

Typescript:

1. Меньше ошибок при разработке
2. Код понятнее и лучше читается
3. Намного проще поддерживать код
4. Быстрая разработка в продакшене
5. Хорош для больших проектов

Итог:

1. В больших проектах начинают появляться ошибки и процесс разработки/рефакторинга усложняется
2. Typescript добавляет типизацию к JS. Подсвечивает большинство ошибок JS
3. Позволяет очень гибко работать с типизацией. Создавать типы на лету.
4. Упрощает поддержку кода на проектах
5. Упрощает жизнь!

Список литературы

1. https://basarat.gitbooks.io/typescript/content/ - книга, которую пишет сообщество. Самая актуальная по языку
2. https://www.typescriptlang.org/ - официальный сайт
3. https://github.com/microsoft/TypeScript - сам проект

LinkedIn: www.linkedin.com/in/nikita-starichenko/ Telegram: @nikita_starichenko