Take a TypeScript

Денис Малиночкин

@mrmlnc

@mrmlnc

Содержание

Вернуться
к содержанию

Современный JavaScript

Современный JavaScript

- Развиваемый
- Поддерживаемый
- Простой (относительно)
- Перспективный
- Типизированный

Современный JavaScript

- Развиваемый
- Поддерживаемый
- Простой (относительно)
- Перспективный
- Типизированный

Современный JavaScript

- На фронтенде
- На бекенде
- Десктопные приложения (Electron, NW.js, React Native, …)
- Мобильные приложения (React Native, Ionic, NativeScript, …)
- Встраиваемые системы

JavaScript занимает значимое место в современном мире

Вернуться
к содержанию

Проблемы в JavaScript

Мы используем JavaScript не так, как это предполагалось в самом начале

Не для больших проектов и команд

Поэтому …

- Создаём тысячи поделок, которые помогают решать нам задачи
- Много мест, где мы что-то делаем не так
- Сложность кода проекта растёт с каждым рабочим днём
- Код-ревью становится сложнее с каждым новым разработчиком и каждым рабочим днём

Рефакторинг на грани

Рефакторинг на грани

- Изменили тело функции
- Забыли поменять JSDoc*
- Следующий человек сделал допущение на основе JSDoc
- Получили баг
- LSR (Live Site Review)

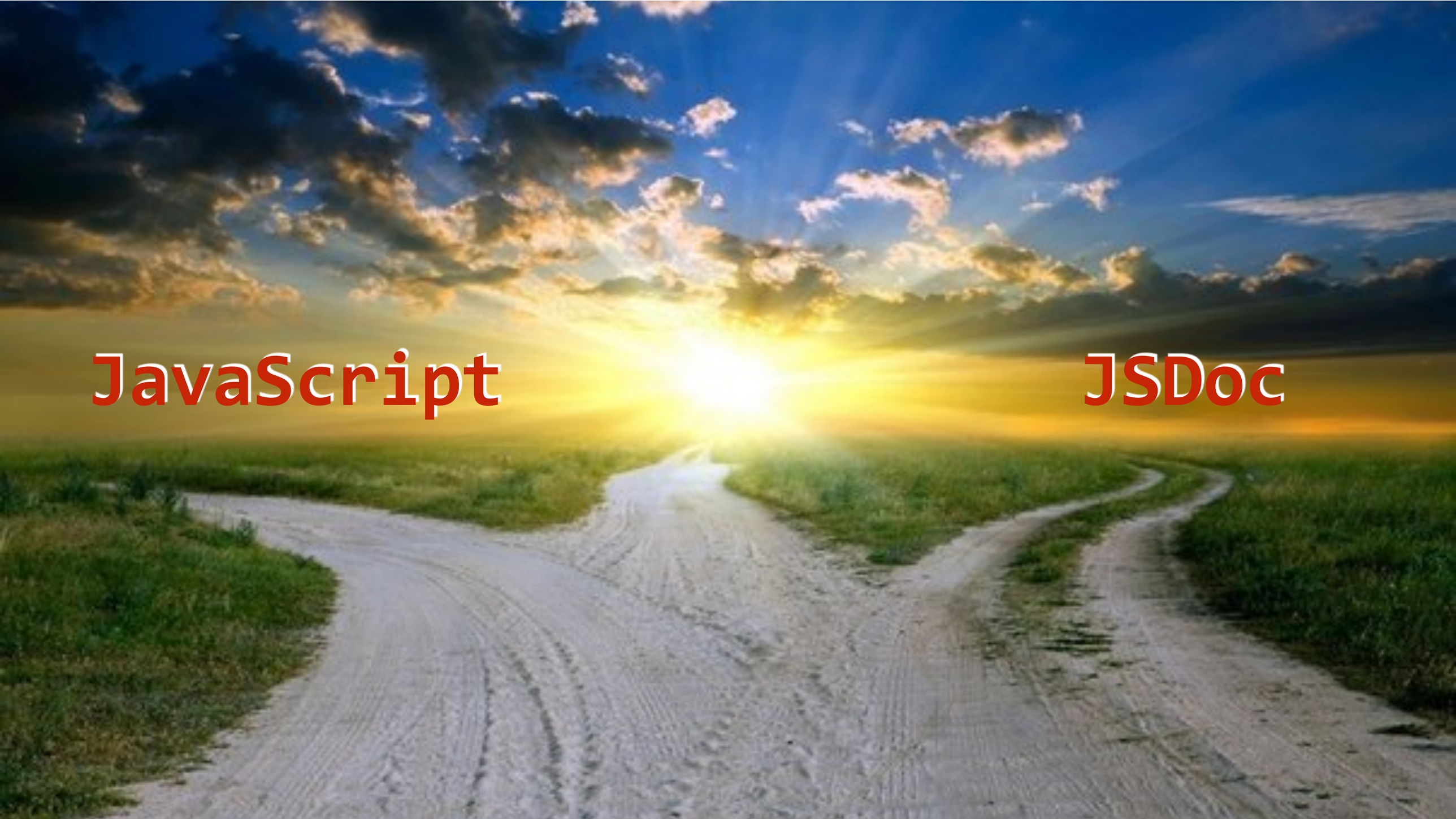
* Если JSDoc написан

Рефакторинг на грани

- Изменили типы принимаемых аргументов
- Забыли про одно место вызова функции
- Получили баг
- LSR (Live Site Review)

Документация отделена от имплементации

JSDoc — костыль сбоку, а не часть языка


JavaScript
JSDoc

Мы **люди**

А людям свойственно

- Ошибаться
- Забывать
  - Наша память не безгранична
  - Мы можем держать во внимании 7 ± 2 объектов
- Откладывать на потом


Всё ! Больше провода не грызу !

Жонглирование типами

В JavaScript есть типы

- Boolean
- Number
- String
- Null
- Undefined
- Object
- Symbol

Жонглирование типами

```
> (![] + []) → 'false'
> '10' + 10 → '1010'
> 'a' + {} → 'a[object Object]'
```


Q W
THIGHS
0.2 meters
O P
CALVES

А потом…

```
› undefined is not a function
› cannot read property 'length' of undefined
```

Вернуться
к содержанию

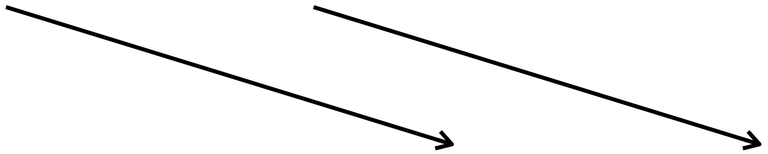
Статическая типизация

Статическая типизация

Аннотация типов в момент написания кода

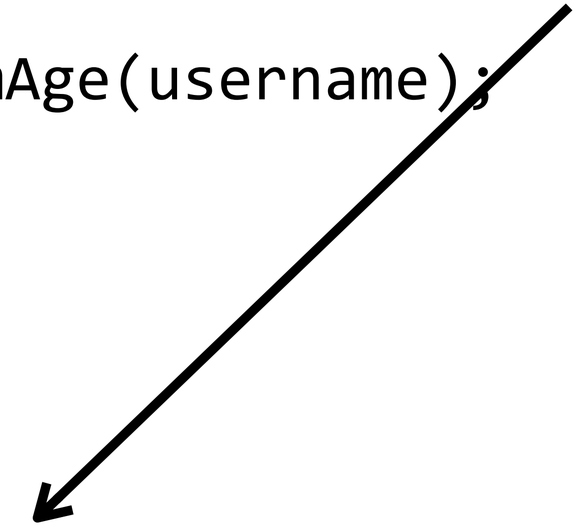
```
function getUserAge(username: string): number {
    return randomAge(username);
}
```

Аннотация типов



```
function getUserAge(username: string): number {
    return randomAge(username);
}
```

```
function getUserAge(username: string): number {
    return randomAge(username);
}
```



```
getUserAge(123);
```

❌

```
getUserAge('mrmlnc');
```

👍

```
// Error ! Argument of type '123' is not assignable to parameter of type 'string'.
```

Передавать числовое значение вместо строки, очевидно, плохая идея.

Статическая типизация определяет лишь то, когда в вашей программе «появляется» знание о типах и ничего более

Статическая типизация

Сильная / Слабая

Сильная типизация

Сильная типизация *(строгая)* требует явного преобразования типов и запрещает автоматические преобразования в

```
userAge = '1' + 1

// TypeError: cannot concatenate 'str' and 'int' objects
```

Слабая типизация

Слабая типизация *(нестрогая)* позволяет делать преобразования типов автоматически.

```
const userAge = '1' + 1; // 11
```

Статическая типизация

Явная / Неявная

Явная типизация

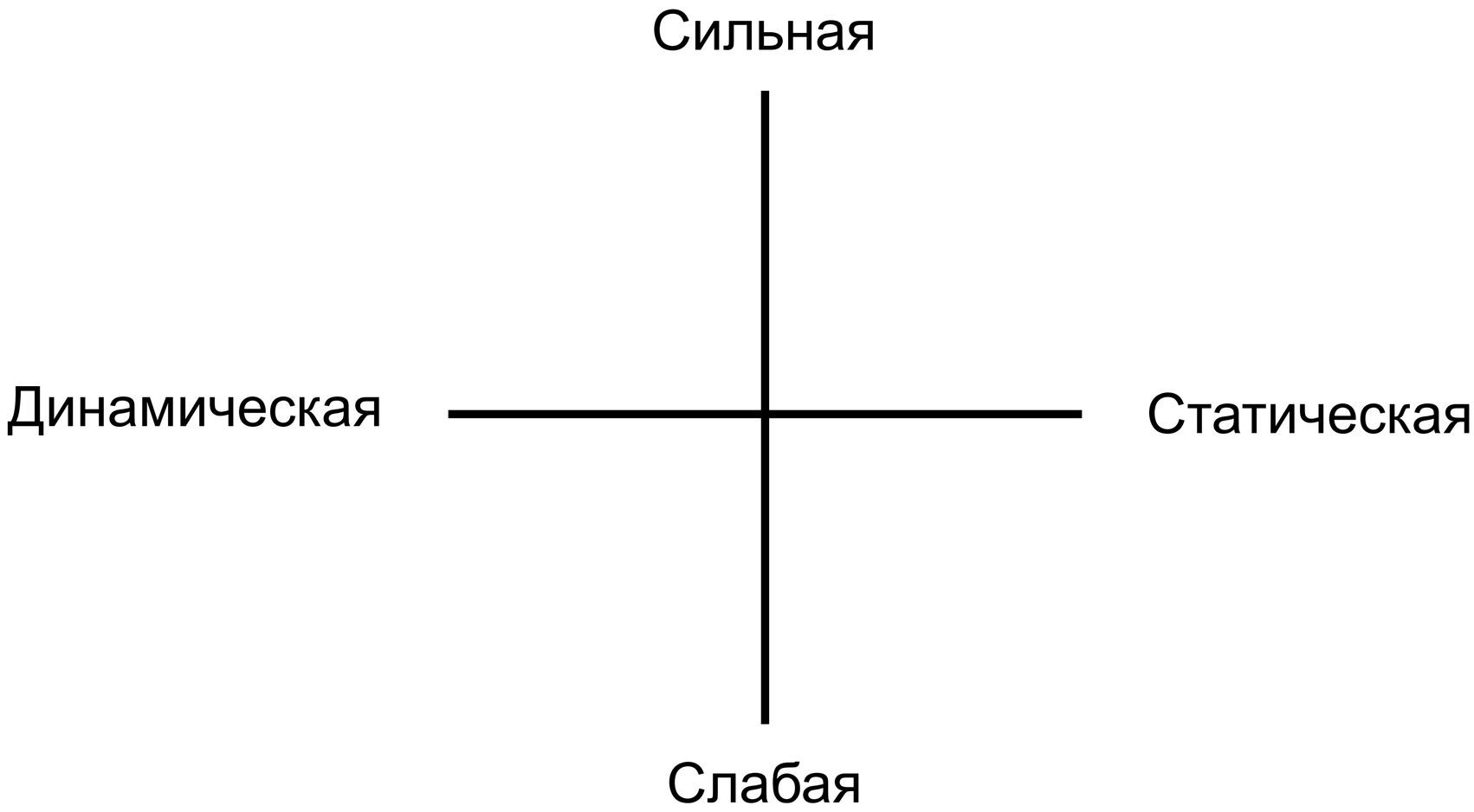
> **Явная типизация** *требует* повсеместного указания типов.

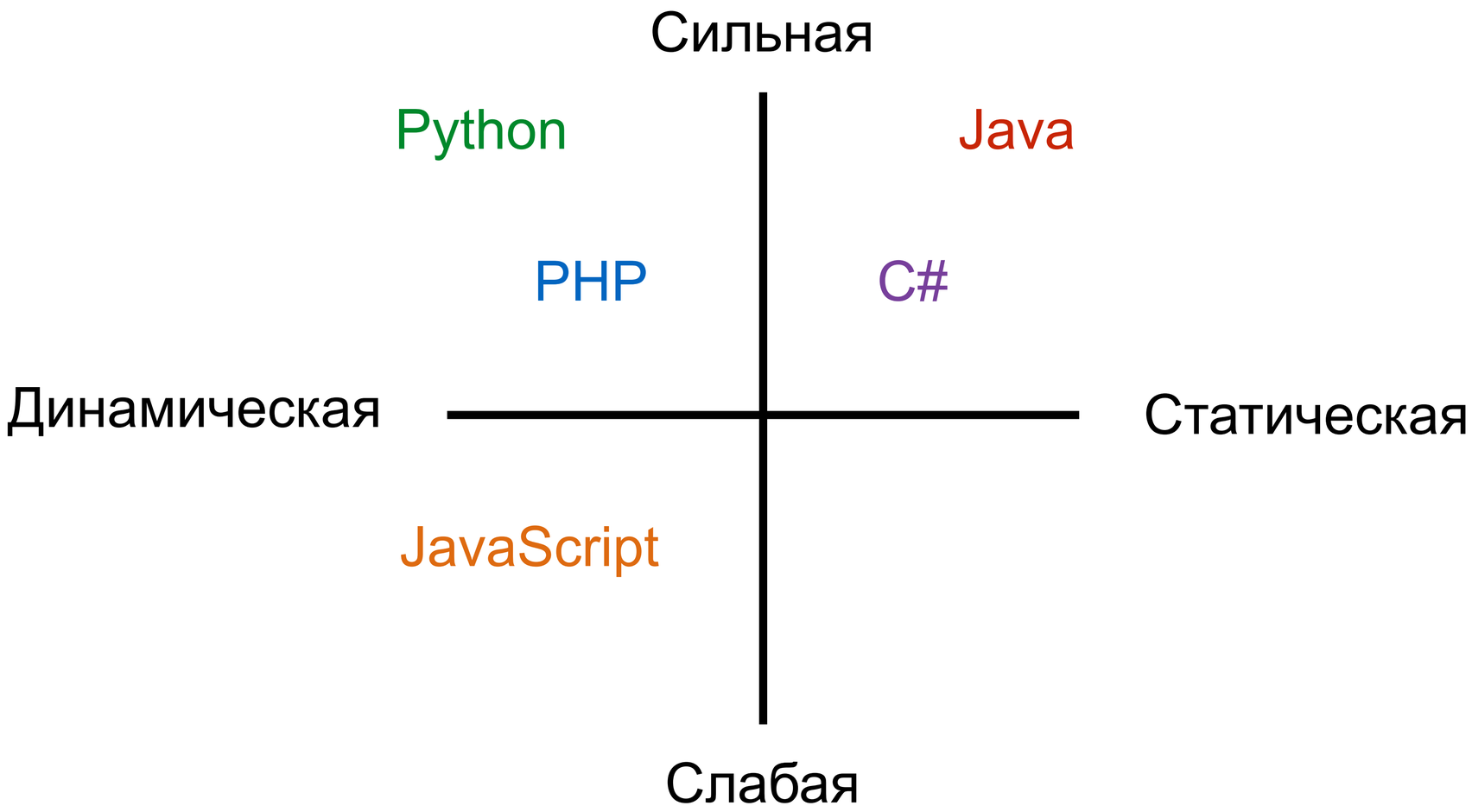
```
const userAge: number = 1;
```

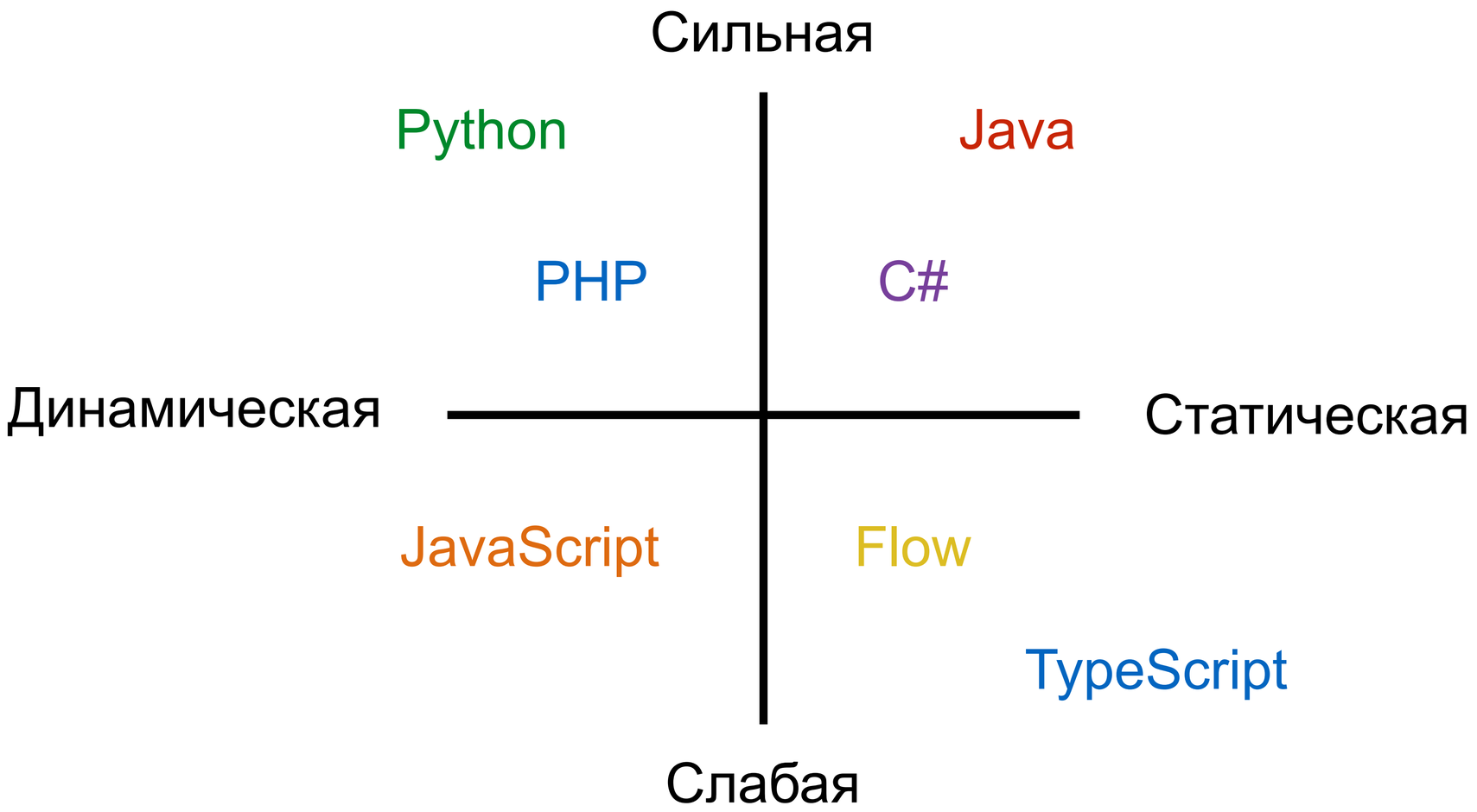
Неявная типизация

> **Неявная типизация** *не требует* повсеместного указания типов.

```
const userAge = 1;
```


Сильная
Динамическая
Статическая
Слабая


Сильная
Python
Java
PHP
C#
Динамическая
Статическая
JavaScript
Слабая


Сильная
Python
Java
PHP
C#
Динамическая
Статическая
JavaScript
Flow
TypeScript
Слабая

Статическая типизация

Возможности

Статическая типизация позволяет

- Выявлять ошибки несоответствия типов
- Создавать абстракции и прототипировать до написания кода
- Бесплатно получить самодокументирование кода
- Дополнить возможности вашей IDE (разумный IntelliSense)
- Упростить рефакторинг кода
- Упростить дебаггинг кода
- Описывать и заключать контракты между всеми разработчиками

Статическая типизация – не серебряная пуля от всех ваших проблем

Статическая типизация **не** позволяет

- Избавиться от тестов
- Избавиться от багов
- Избавиться от необходимости писать код и ходить на работу

Статическая типизация

Плюсы / Минусы

Плюсы статической типизации

- Можно найти ошибки соответствия ожидаемых/реальных типов на этапе компиляции программы
- Меньше шансов совершить ошибку, используя то, что не доступно текущему типу или блоку инструкций
- В больших проектах это даёт прирост к скорости разработки и надёжности кодовой базы
- Очень приятный рефакторинг кода
- Есть небольшой прирост производительности (сомнительно)

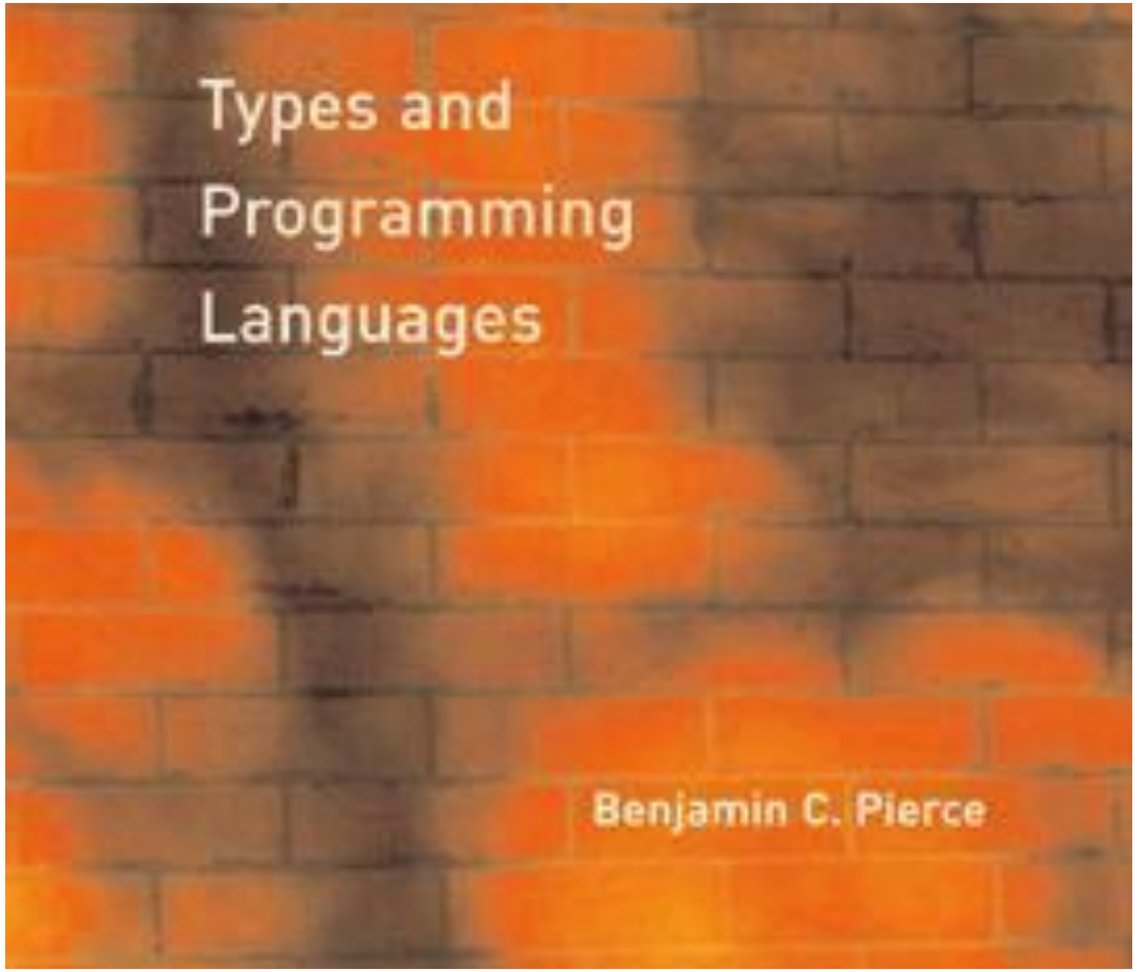
Минусы статической типизации

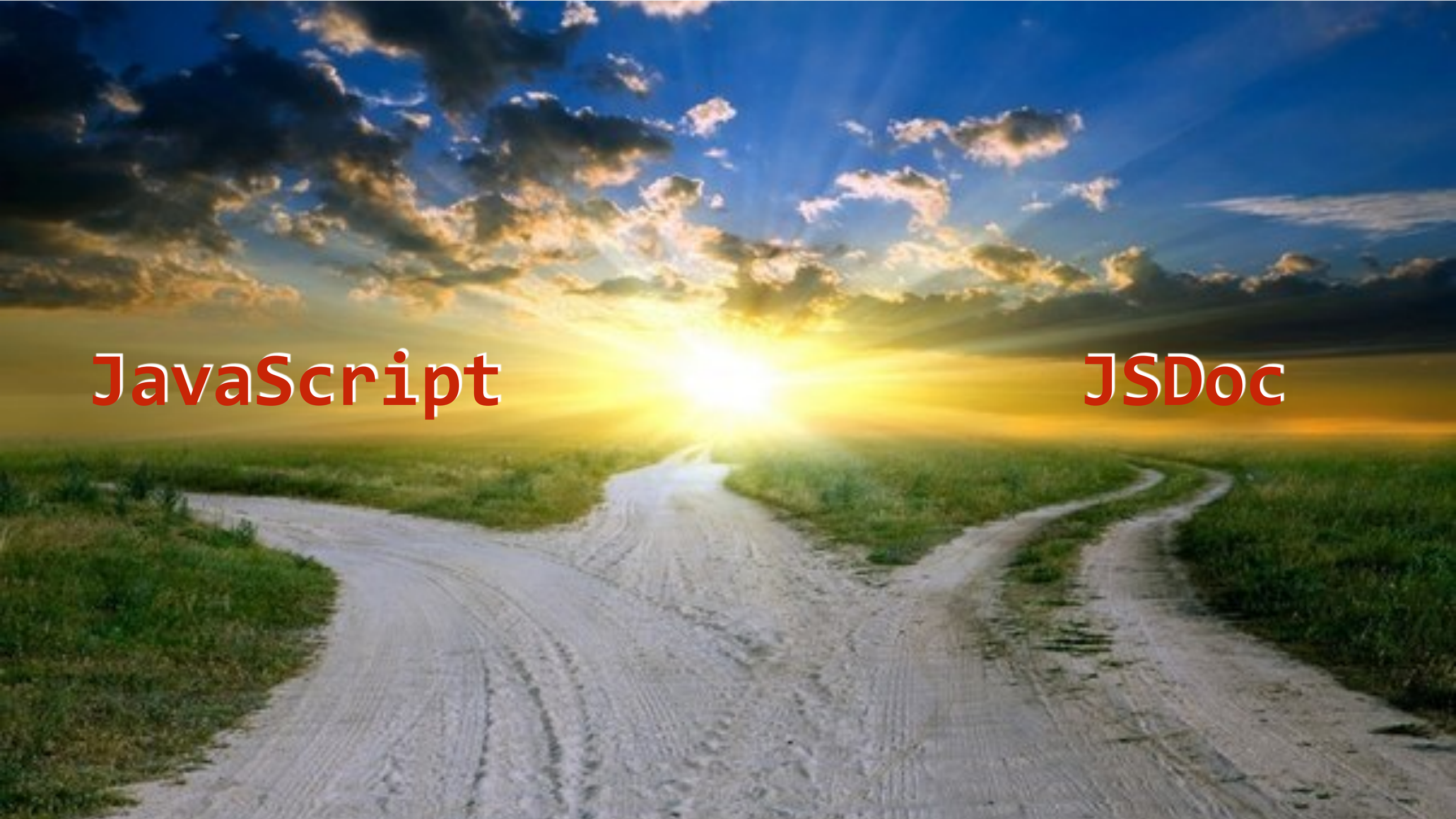
- Необходимость покрывать код метаинформацией (типами)
- Иногда, самый безобидный рефакторинг может задевать значимое число мест кодовой базы

Литература

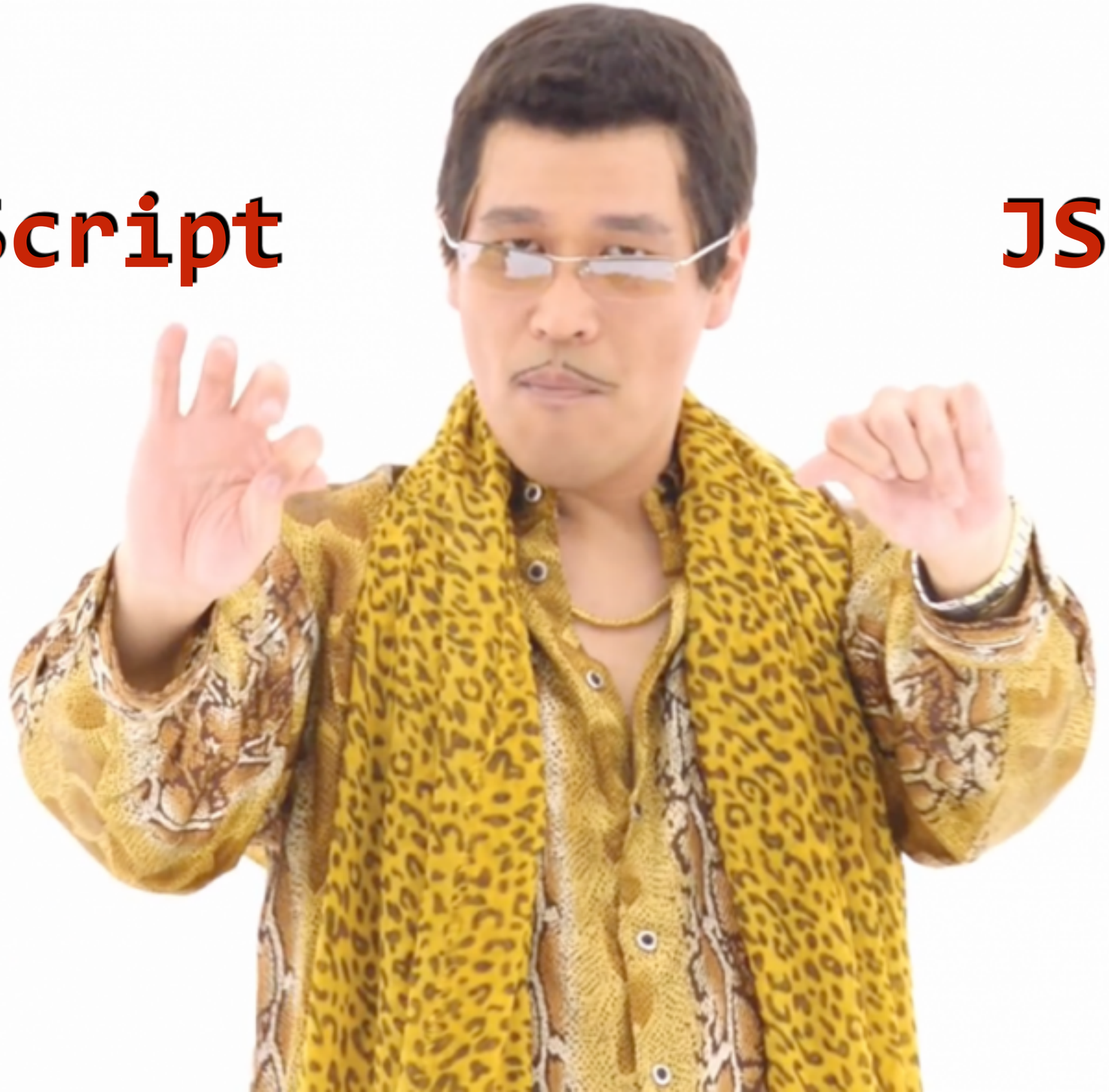
Types and Programming Languages

Benjamin C. Priece



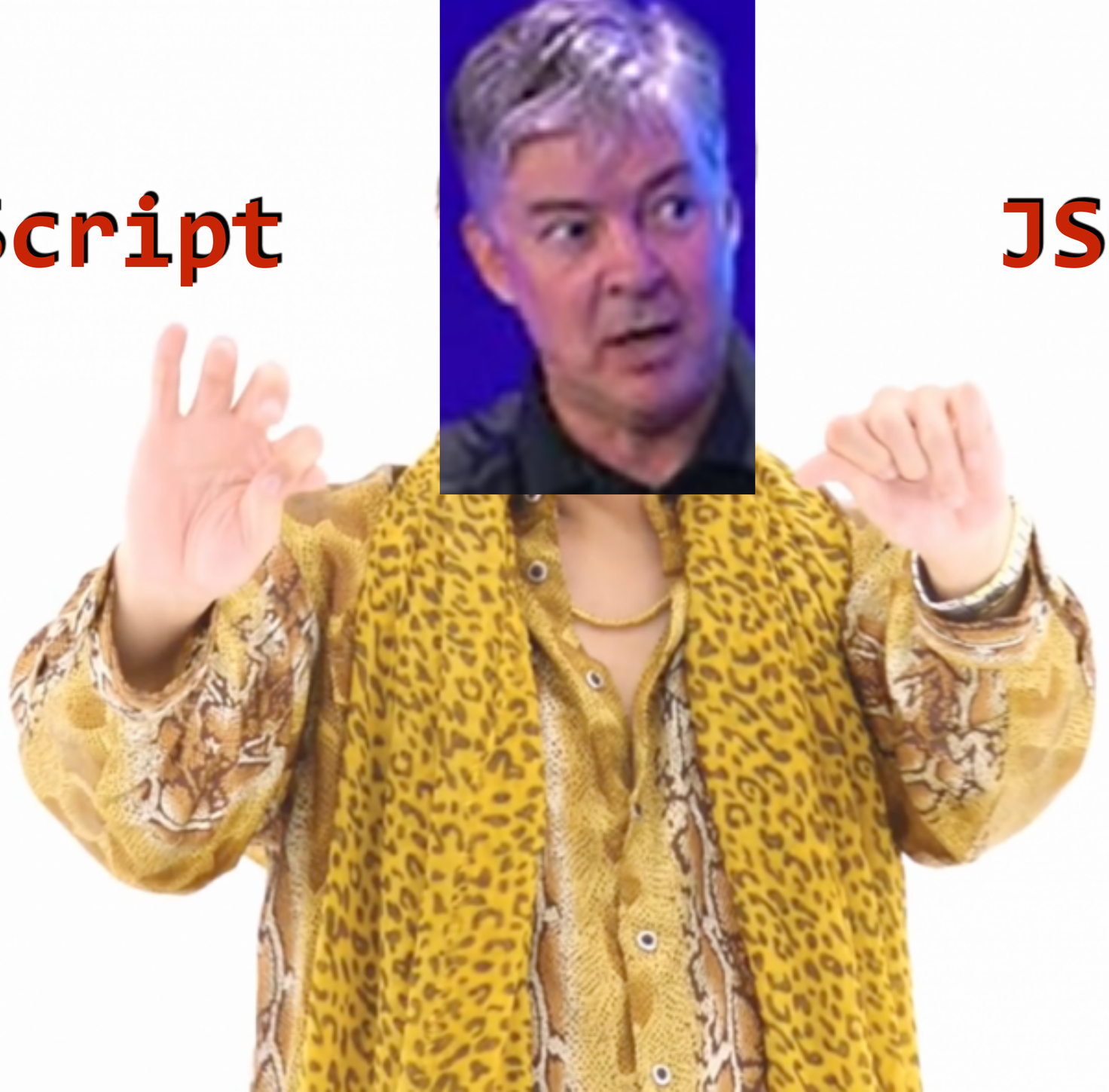

JavaScript
JSDoc

JavaScript



JSDoc

JavaScript



JSDoc

Вернуться
к содержанию

TypeScript

TypeScript...

TypeScript – это компилируемое надмножество JavaScript, приносящее опциональную статическую типизацию.

TypeScript...

TypeScript – это **компилируемое** надмножество JavaScript, приносящее опциональную статическую типизацию.

TypeScript...

TypeScript – это **компилируемое надмножество JavaScript**, приносящее опциональную статическую типизацию.

TypeScript...

TypeScript – это **компилируемое надмножество JavaScript**, приносящее **опциональную статическую типизацию**.

Хочешь типы


Хочешь типы – ИСПОЛЬЗУЕШЬ

```
function getUserAge(username: string): number {
    return randomAge(username);
}
```

```
function getUserAge(username) {
    return randomAge(username);
}
```

Знание о типах полностью пропадает после компиляции TypeScript в JavaScript

TypeScript оперирует структурами

Структуры

```
interface INinja { field: string; }
```

Структуры

```
interface INinja { field: string; }

function getNinjaField(ninja: INinja): string { // something }
```

Структуры

```
interface INinja { field: string; }

function getNinjaField(ninja: INinja): string { // something }

class Ninja implements INinja {
    public get field() { return 'value'; }
};

const ninja: INinja = { field: 'value' };
const ninja2 = new Ninja();

getNinjaField(ninja);
getNinjaField(ninja2);
```

Структуры

```
interface INinja { field: string; }

function getNinjaField(ninja: INinja): string { // something }

class Ninja implements INinja {
    public get field() { return 'value'; }
};

const ninja: INinja = { field: 'value' };
const ninja2 = new Ninja();

getNinjaField(ninja);
getNinjaField(ninja2);
```

Компиляция в JavaScript

```
› $ npm i (-g) typescript
› $ touch tsconfig.json
› $ tsc -p .
› $ node ./out/zip-bomb.js
```

Устанавливаем, конфигурируем, компилируем и запускаем – всё очень просто.

Содержимое tsconfig.json

```
{
    "compilerOptions": {
        "target"  : "es5",
        "module"  : "commonjs",
        "rootDir" : "src",
        "outDir"  : "out"
    }
}
```

Содержимое tsconfig.json

```
{
    "compilerOptions": {
        "target"  : "es5",
        "module"  : "commonjs",
        "rootDir" : "src",
        "outDir"  : "out"
    }
}
```

Содержимое tsconfig.json

```
{
    "compilerOptions": {
        "target"  : "es5",
        "module"  : "commonjs",
        "rootDir" : "src",
        "outDir"  : "out"
    }
}
```

Содержимое tsconfig.json

```
{
    "compilerOptions": {
        "target"  : "es5",
        "module"  : "commonjs",
        "rootDir" : "src",
        "outDir"  : "out"
    }
}
```

JS → TS

Проблема

```
/**
 * @param {number} a - Описание параметра.
 * @param {number} b - Описание параметра.
 * @return {number}
 */
const sum = (a, b) => a + b;
```

Проблема

```
/**
 * @param {number} a - Описание параметра.
 * @param {number} b - Описание параметра.
 * @return {number}
 */
const sum = (a, b) => a + `${b}`;
```

Проблема

```
/**
 * @param {number} a - Описание параметра.
 * @param {number} b - Описание параметра.
 * @return {number}
 */
const sum = (a, b) => a + `${b}`;
```

TypeScript

```
/**
 * @param a - Описание параметра.
 * @param b - Описание параметра.
 */
const sum = (a: number, b: number): number => a + b;
```

TypeScript

```ts
/**
 * @param a - Описание параметра.
 * @param b - Описание параметра.
 */
const sum = (a: number, b: number): number => a + b;
```

JSDoc никуда не уходит

JSDoc теперь **описывает параметры**

TS → JS

TypeScript

```
/**
 * @param a - Описание параметра.
 * @param b - Описание параметра.
 */
const sum = (a: number, b: number): number => a + b;
```

TypeScript после компиляции (ES2015)

```
/**
 * @param a - Описание параметра.
 * @param b - Описание параметра.
 */
const sum = (a, b) => a + b;
```

TypeScript после компиляции (ES5)

```
/**
 * @param a - Описание параметра.
 * @param b - Описание параметра.
 */
var sum = function (a, b) { return a + b };
```

Скомпилированный JS **максимально похож на исходники**

Class → ES5

```
var MySuperClass = /** @class */ (function () {
    function MySuperClass() {
    }
    return MySuperClass;
}());
```

Привет, Babel

```
function _classCallCheck(instance, Constructor) { if (!
(instance instanceof Constructor)) { throw new
TypeError("Cannot call a class as a function"); } }

var MySuperClass = function MySuperClass() {
  _classCallCheck(this, MySuperClass);
};
```

Магия...

Синтаксическое понижение кода (транспиляция)

(!) Если синтаксической конструкции нет в указанной спецификации, то транспилируем код до её уровня.

Синтаксическое понижение кода

- let/const
- Arrow Functions
- Default Parameters
- Class
- Destructuring
- Spread Operator
- Rest Parameters
- for…of
- Template Strings
- Async/Await
- ES2015 modules
- Decorators*

Не полифилит ранее не существовавшие методы (`Object.assign`)

Сверхширокая поддержка IDE

Ядро

- Парсер, пре-процессор, связыватель, контроллер типов, эмиттер

Автономный компилятор

- API компилятора без привязки к окружению

Языковой сервис

- Инструментарий для IDE (Refactor, Navigation, Hover, Autocomplete)

Прочие слои

HOW STANDARDS PROLIFERATE:

(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)

SITUATION: THERE ARE 14 COMPETING STANDARDS.

14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.

SOON:

SITUATION: THERE ARE 15 COMPETING STANDARDS.

Language Server Protocol

Microsoft, Eclipse, IBM, Red Hat, Codenvy

Language Server Protocol

Стандартизирует то, как редактор может общаться с языковым сервером и то, как этот сервер может отвечать редактору на разные запросы.

- Autocomplete
- Hover
- Refactor
- Go To
- …

Language Server Protocol

Реализовано

- VS + **VS Code**
- Eclipse IDE + Eclipse Che
- **Atom**
- **Sublime Text 3** (плагин)
- neovim

Ожидается

- Vim (плагин)
- Emacs

Что-то там

- JetBrains

C#

Наследие от C#

- Types
- Interfaces
- Generics
- Enum
- Abstract classes
- Function Overloads (не для всех случаев)

А вот это важно понимать

- (!) Types
- (!) Interfaces
- (!) Generics
- Enum
- Abstract classes
- Function Overloads (не для всех случаев)

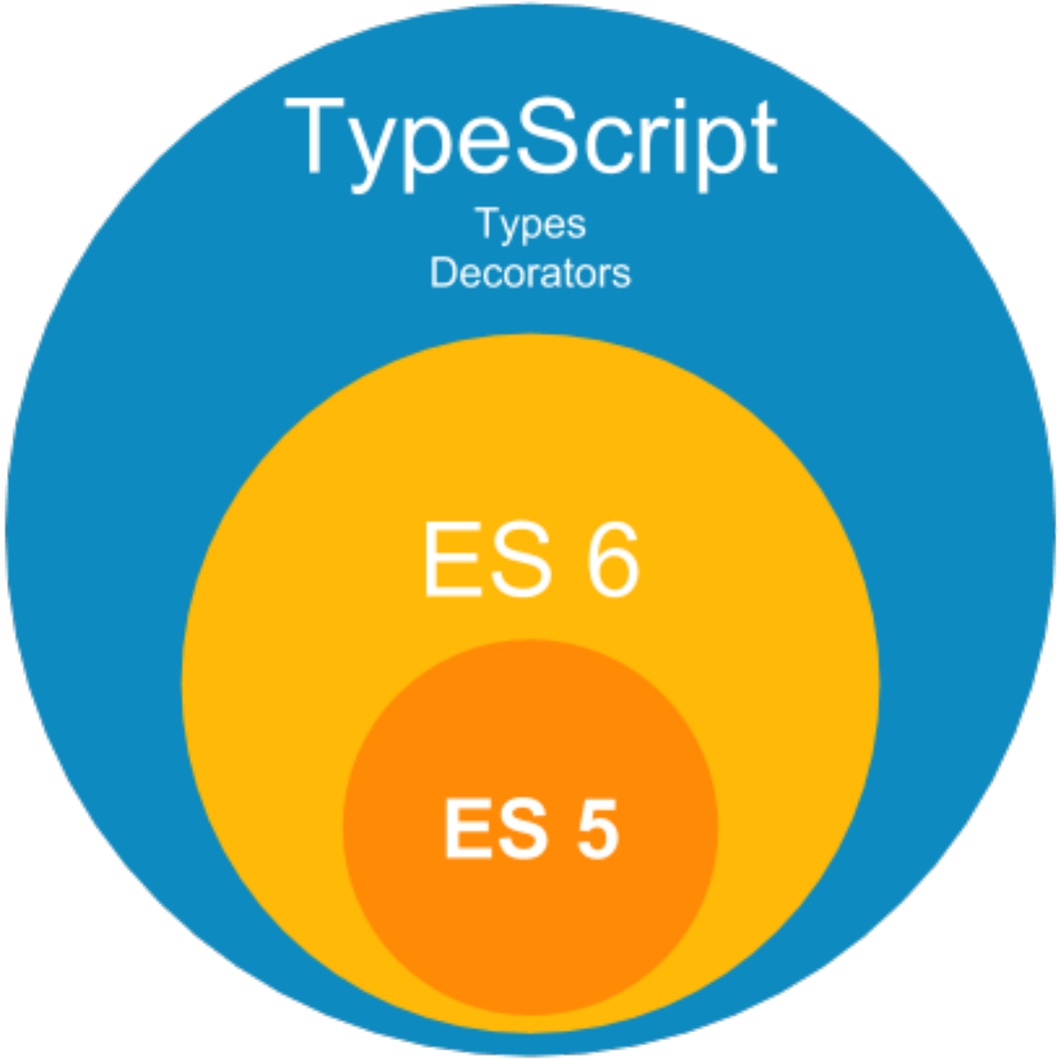
Не то, чтобы реклама, но...

- Введение
- Типы и Структуры
- Обобщённые типы (Generics)

www.canonium.com/category/typescript

TypeScript

- Компиляция в JavaScript
- Статическая типизация
- Структурный язык
- Синтаксическое понижение кода
- Сверхширокая поддержка IDE
- Возможность расширения
- TypeScript – это всё тот же JavaScript



One more thing...

Вы можете использовать TS в JS

tsconfig.json

```
{
    "compilerOptions": {
        "target": "es3",
        "allowJs": true,  // разрешаем компилировать JS
        "checkJs": true   // просим проверять типы в JS
    }
}
```

// @ts-check

Говорим компилятору, что нужно проверять типы в файле

Компилятор проверяет JS файлы

- JSDoc
  - Типичный валидный JSDoc
- @type
  - `/** @type {SomeType} */`

Сделать процесс покрытия кода типами максимально простым

TypeScript

Плюсы / Минусы

Плюсы TypeScript

- Популярное и готовое к продакшену решение
- Проект развивается (стабильный релизный цикл)
- Мощная поддержка комьюнити
- Есть возможность проверять JS на основе JSDoc
- Написан на TypeScript

Минусы TypeScript

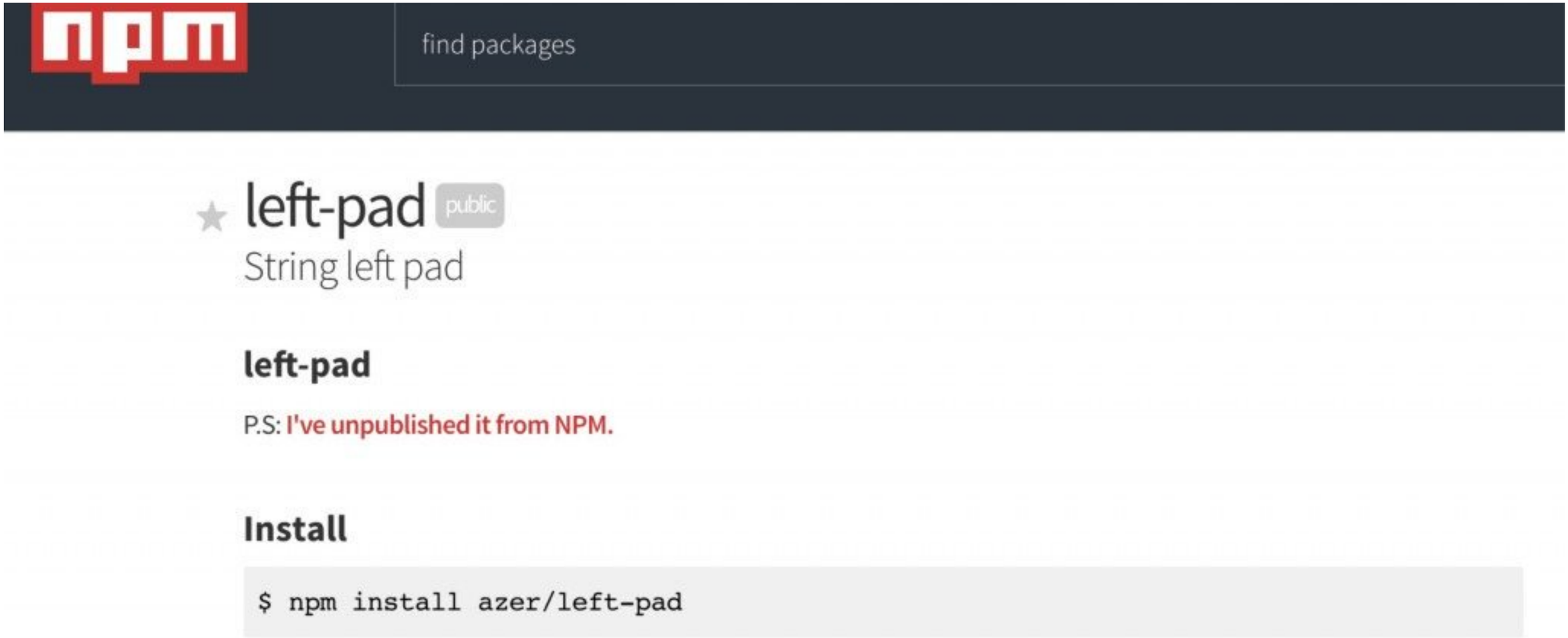
- Нет «живой» спецификации языка – есть анонсы для конечных пользователей
- Нет документации самого компилятора и LSP-слоя
- Местами простые вещи делаются сложно*

* Исправляется

TypeScript

А как быть с наследием JavaScript

Как же я без пакетов?



TypeScript = JavaScript + Types

Declarations Files

.d.ts

Описание конструкций

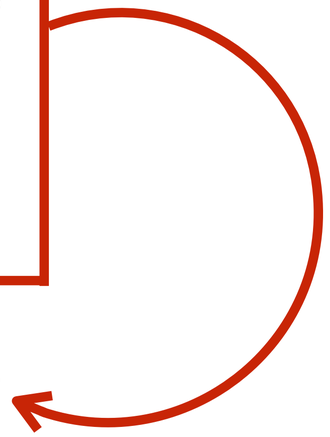
Статическое описание конструкций, а не логика

Пример посложнее

```
declare module "svg-parser" {
    namespace svgParser {
        interface INode {
            name: string;
            attributes: Record<string, string>;
            children: INode[];
            metadata?: string;
        }
        function parse(content: string): INode;
    }
    export = svgParser;
}
```

Пример посложнее

```
declare module "svg-parser" {
    namespace svgParser {

        interface INode {
            name: string;
            attributes: Record<string, string>;
            children: INode[];
            metadata?: string;
        }

        function parse(content: string): INode;
    }
    export = svgParser;
```



Заголовочные файлы позволяют **ограничивать**

Ограничения через `.d.ts`

- Указал целевую спецификацию ES5
  - Забыл про **Promise**
  - Забыл про **прочие новинки**

- Добавил в проект полифиллы/понифиллы
  - Разрешил использовать Promise
  - Разрешил что-то ещё

@types/*

Задекларированные типы для множества npm-пакетов

Вернуться
к содержанию

Два слайда про Flow

Тип логики

TypeScript

- **Completness** (целостность)
- Анализ ошибок, которые случатся в рантайме
- Типы дополняют код

Flow

- **Soundness** (надёжность)
- Анализ возможных ошибок, которые могут случиться в рантайме
- Вы доказываете компилятору, что вы правы

Просто Flow

```
interface IObject {
    value?: string;
}

const obj: IObject = {};

function showStringLength(x: string) {
    console.log(x.length);
}

if (typeof obj.value === 'string') {
    showStringLength(obj.value);
    showStringLength(obj.value); // A obj тот же?  <--- Добавь IF
}
```

Вернуться
к содержанию

Типы и структуры данных

Типы данных

Типы данных в JavaScript

- boolean
- null
- undefined
- number
- string
- Symbol
- Object

Типы данных в TypeScript

- any
- void
- never
- Тип строкового и числового литерала
- object и Object
- Тип литерала объекта
- Полиморфный тип this или F-ограниченный полиморфизм

Типы данных в TypeScript

- any
- void
- never
- Тип строкового и числового литерала
- ~~object и Object~~
- Тип литерала объекта
- Полиморфный тип this или F-ограниченный полиморфизм

Типы данных в TypeScript

- any
- void
- never
- Тип строкового и числового литерала
- ~~object и Object~~
- Тип литерала объекта
- ~~Полиморфный тип this или F-ограниченный полиморфизм~~

Поехали

any

Определение типа «any»

Определение

Тип, предполагающий под собой любой тип.

Зачем нужен?

- Совместимость с JavaScript
- Отображение того, что мы не знаем тип

Когда применять тип?

- Никогда

Примеры для типа «any»

```
const a: any = 1;
const b: any = {};
```

```
import glob = require('glob');

const entries = glob.sync('**/*');
```

Примеры для типа «any»

```ts
const a: any = 1;
const b: any = {};
```

```ts
import glob = require('glob');

const entries = glob.sync('**/*');  // Type: ???
```

Примеры для типа «any»

```
const a: any = 1;
const b: any = {};
```

```
import glob = require('glob');

const entries = glob.sync('**/*');  // Type: any
```

Приписание типов

Что делать, если **я** умнее компилятора?

Подсказки компилятору

```
const a: any = 'this is a string';


// Компилятор: ну ОК – тебе лучше знать
const b: number = a.qwe();
```

Подсказки компилятору

```
const a: any = 'this is a string';


// Компилятор: ну ОК - тебе лучше знать
const b: number = a.qwe();  // 👍
```

Подсказки компилятору

```
const a: any = 'this is a string';


// Компилятор: ну ОК – тебе лучше знать
const b: number = a.qwe();  // ❌ Runtime error
```

Подсказки компилятору

```
const a: any = 'this is a string';


// Компилятор: ну ОК – тебе лучше знать
const b: number = a.qwe();


// Компилятор: О, а, оказывается, строка
const c: number = (a as string).qwe();
```

Подсказки компилятору

```
const a: any = 'this is a string';


// Компилятор: ну ОК – тебе лучше знать
const b: number = a.qwe();


// Компилятор: О, а, оказывается, строка
const c: number = (a as string).qwe(); // 👍  Compilation error
```

Type assertions

```
const broken = [].concat('dir');

// Error: [ts] Argument of type '"dir"' is not assignable to
parameter of type 'ReadonlyArray<never>'.
```

```
const success = ([] as string[]).concat('dir');  // 👍
```

`void`

Определение типа «void»

Определение

Тип, предполагающий под собой ничего.

Зачем нужен?

- Возможность сказать, что эта функция должна возвращать ничего

Примеры для типа «void»

```
import mq = require('mq');

function stopQueue(): void {
    mq.stop();
}
```

```
const result = stopQueue();
```

Примеры для типа «void»

```
import mq = require('mq');

function stopQueue(): void {
    mq.stop();
}
```

```
const result = stopQueue();  // Type: void
```

null/undefined

Тип «null» и «undefined»

null

- Включён во все типы (`null`, `undefined`, `number`, `object`, `Object`…)

undefined

- Включён во все типы (`undefined`, `null`, `number`, `object`, `Object`…)

Тип «null» и «undefined»

null

- Включён во все типы (`null`, **`undefined`**, `number`, `object`, `Object`…)

undefined

- Включён во все типы (`undefined`, **`null`**, `number`, `object`, `Object`…)

--strictNullChecks

- `undefined` и `null` – **независимые** типы

Примеры «null»

```
const a: null = null;

const b: string = null;
```

Примеры «null» (--strictNullChecks)

```
const a: null = null;

const b: string = null;  // Error
```

Примеры «undefined»

```
const a: undefined = undefined;

const b: string = undefined;
```

Примеры «undefined» (--strictNullChecks)

```
const a: undefined = undefined;

const b: string = undefined;  // Error
```

What?

Как так-то, а? Я хочу **undefined**!

union

Тип «union»

Возможность указать два и более типа

```
let a: number | string = 1;
let b: number | undefined = undefined;
let c: number | null = null;
```

Тип «union»

Возможность указать два и более типа

```
let a: number | string = 1;
let b: number | undefined = undefined;
let c: number | null = null;
```

Тип «union»

```
(string | number)[]
```

```
string[] | number[]
```

Тип «union»

```
(string | number)[]
```

```
['string', 1]
```

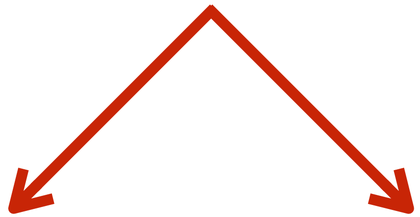
```
string[] | number[]
```

Тип «union»

```
(string | number)[]
```

```
['string', 1]
```

```
string[] | number[]
```



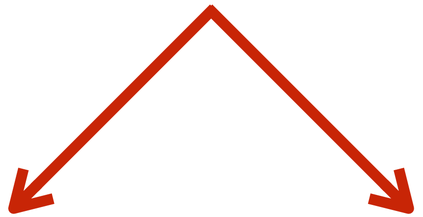
```
string[]    number[]
```

Тип «union»

```
(string | number)[]
```

```
['string', 1]
```

```
string[] | number[]
```



```
string[]    number[]
```

```
['string']     [1]
```

Свой личный тип

'pikachu'

String literal

numeric, object

Тип строкового литерала

Возможность указать заранее известный набор строк, как типы

```
const a: 'pickachu' = 'pikachu';
const b: 'pikachu' | 'charmander ' = 'charmander';


const c: 'charmander' = 'pikachu';
```

Числовой литерал и литерал объекта

```ts
const a: -1 | 0 | 1 = 1;
```

```ts
const b: { a: 1 } | { a: 'abc' } = { a: 'abc' };
```

Числовой литерал и литерал объекта

```ts
const a: -1 | 0 | 1 = 1;
```

```ts
const b: { a: 1 } | { a: 'abc' } = { a: 'abc' };
```

never

Тип «never»

Тип не простой, а *никогда* недостижимый и «богатый».

- Исключение
- Бесконечный цикл
- Недостижимый код

Бесконечный цикл

```
function eventLoop(): never {
    while (true) {
        console.log('tick');
    }
}
```

Недостижимый код

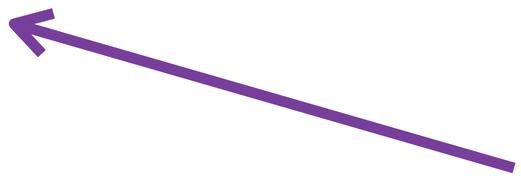
```
function throwCriminalError(): never {
    throw new Error('Mama ama criminal!');
}
```

Недостижимый код

```
function assertCriminal(login: 'mrmlnc' | 'someone'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError();
}
```

Недостижимый код

```
function assertCriminal(login: 'mrmlnc' | 'someone'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError();
}
```

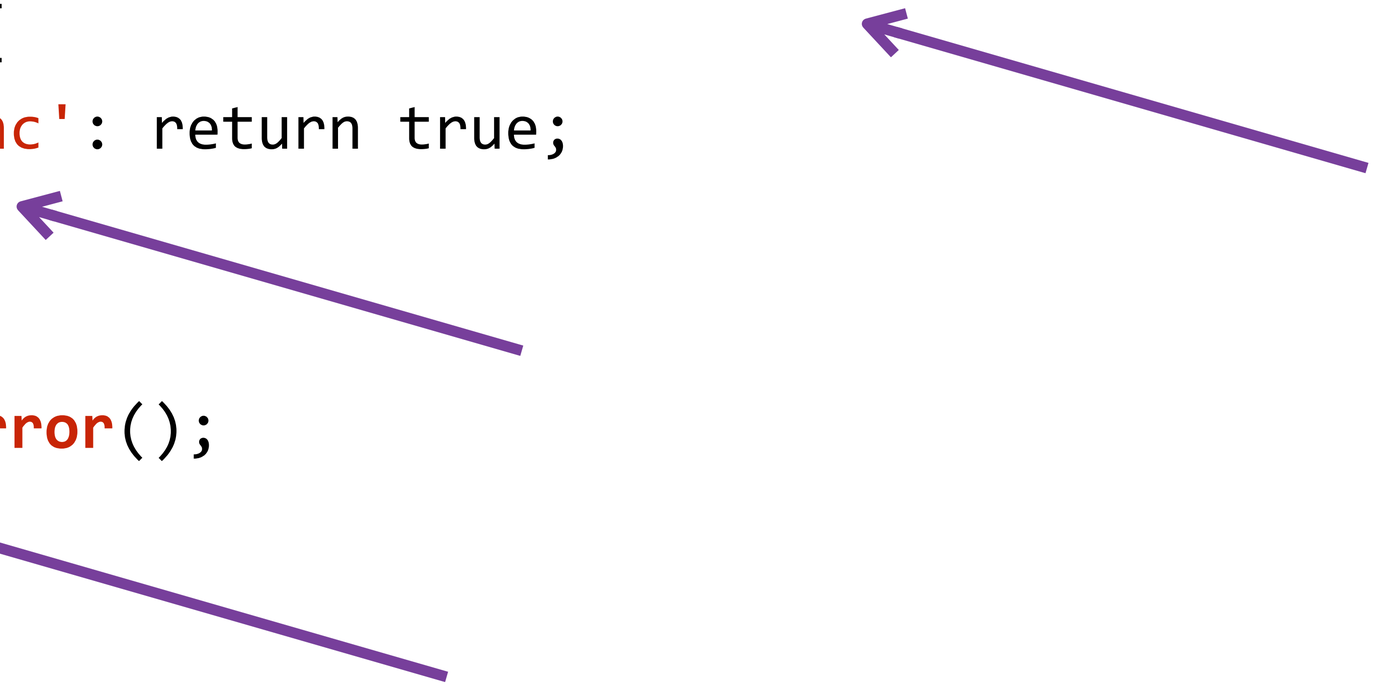


Недостижимый код

```
function assertCriminal(login: 'mrmlnc' | 'someone'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError();
}
```

Недостижимый код

```
function assertCriminal(login: 'mrmlnc' | 'someone'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError();
}
```



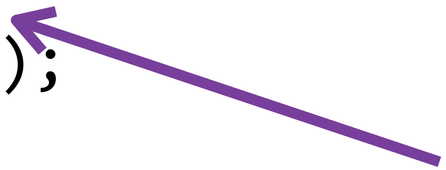
Поможем найти криминал!

Исправляем ошибки

```
function throwCriminalError(login: never): never {
    throw new Error('Mama ama criminal!');
}
```

Исправляем ошибки

```
function throwCriminalError(login: never): never {
    throw new Error('Mama ama criminal!');
}
```

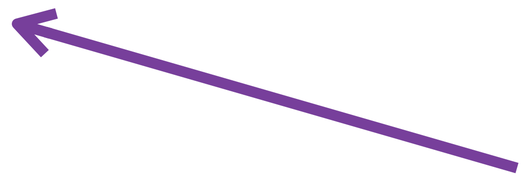


Исправляем ошибки

```
function assertCriminal(login: 'mrmlnc' | 'pikachu'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError();
}
```

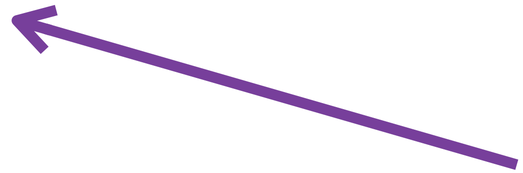
Исправляем ошибки

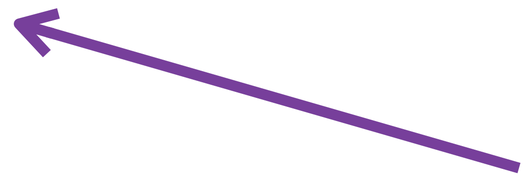
```
function assertCriminal(login: 'mrmlnc' | 'pikachu'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError();
}
```



Исправляем ошибки

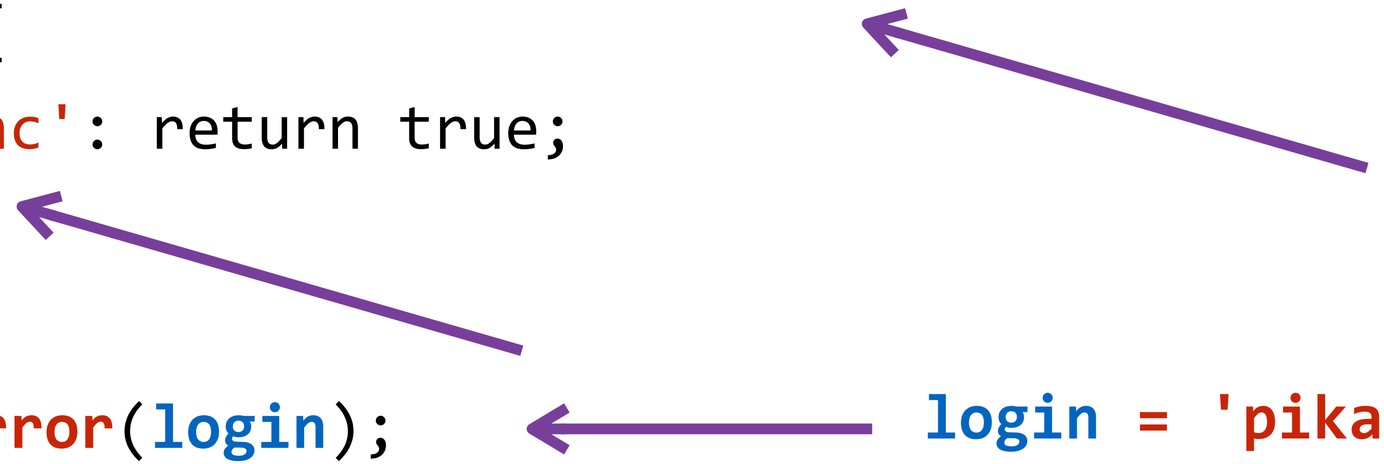
```
function assertCriminal(login: 'mrmlnc' | 'pikachu'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError();
}
```





Исправляем ошибки

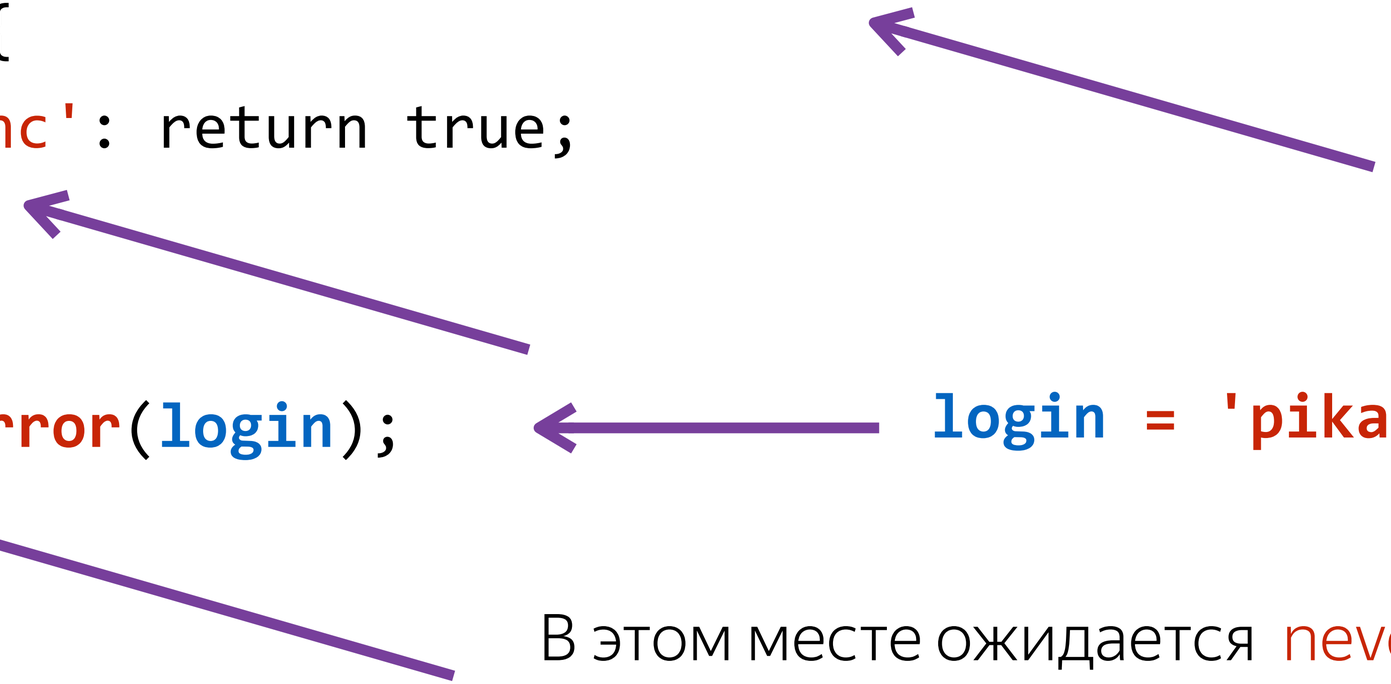
```
function assertCriminal(login: 'mrmlnc' | 'pikachu'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError(login);
}
```



login = 'pikachu'

Исправляем ошибки

```
function assertCriminal(login: 'mrmlnc' | 'pikachu'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError(login);
}
```



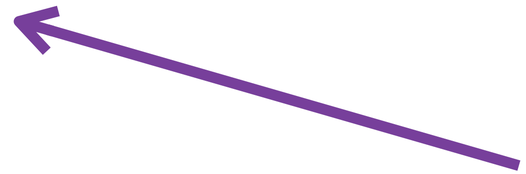
login = 'pikachu'

В этом месте ожидается never,
но сейчас здесь string!

Как починить?

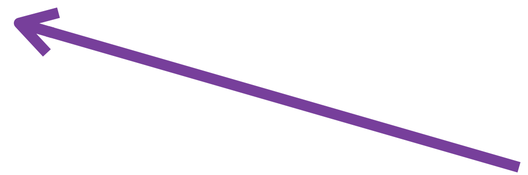
Исправляем ошибки

```
function assertCriminal(login: 'mrmlnc' | 'pikachu'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
    }

    throwCriminalError(login);
}
```



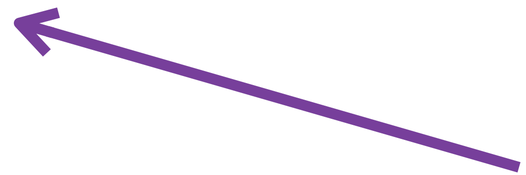
Исправляем ошибки

```
function assertCriminal(login: 'mrmlnc' | 'pikachu'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
        case 'pikachu': return true;
    }

    throwCriminalError(login);
}
```

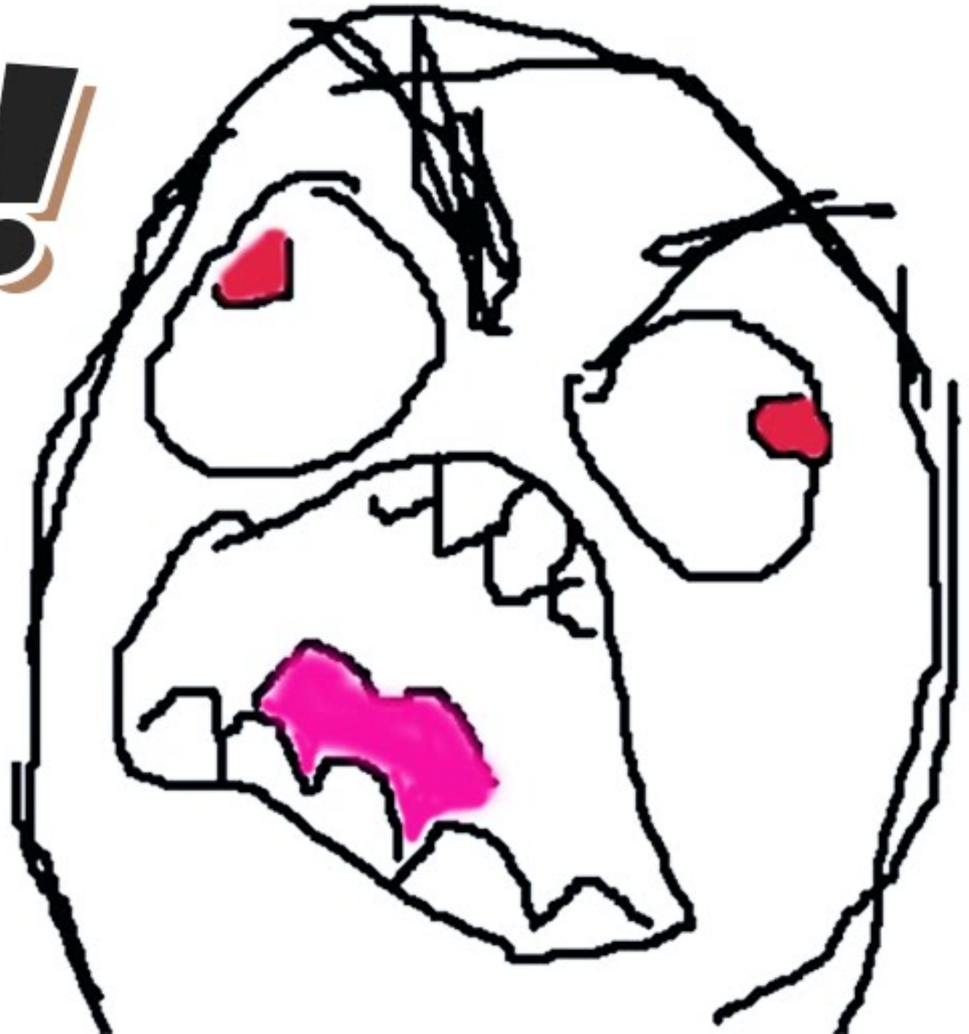


Исправляем ошибки

```
function assertCriminal(login: 'mrmlnc' | 'pikachu'): boolean {
    switch(login) {
        case 'mrmlnc': return true;
        case 'pikachu': return true;
    }

    throwCriminalError(login);  // 👍
}
```



ЧЁ ТАК СЛОЖНО!



Структуры данных

Структуры данных в TypeScript

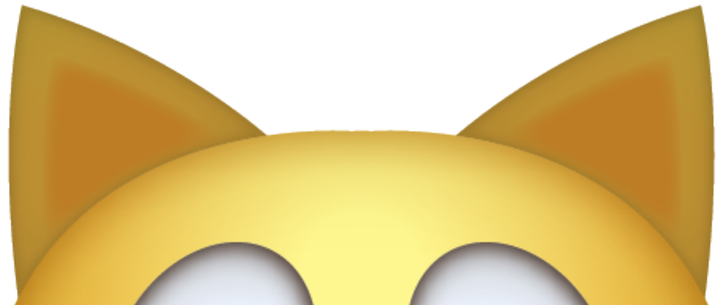
- Объединение (union)
- Пересечение (intersection)
- Перечислители (enum)
- Интерфейс (interface)
- Кортежи (tuple)

Структуры данных в TypeScript

- ~~Объединение (union)~~
- Пересечение (intersection)
- Перечислители (enum)
- Интерфейс (interface)
- Кортежи (tuple)

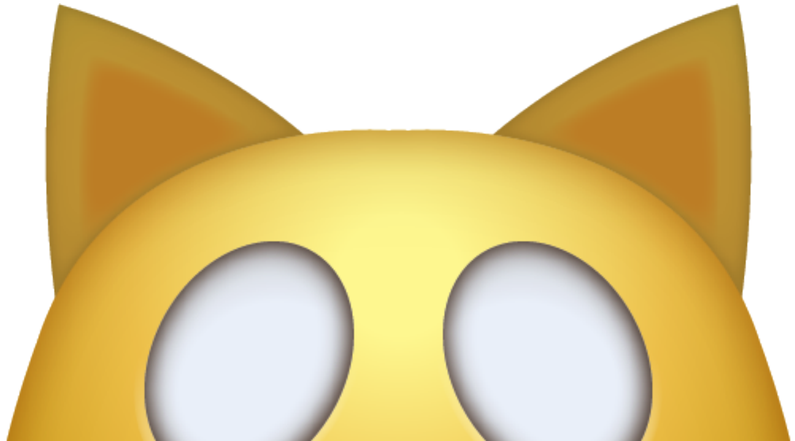
Структуры данных в TypeScript

- ~~Объединение (union)~~
- ~~Пересечение (intersection)~~
- Перечислители (enum)
- Интерфейс (interface)
- Кортежи (tuple)



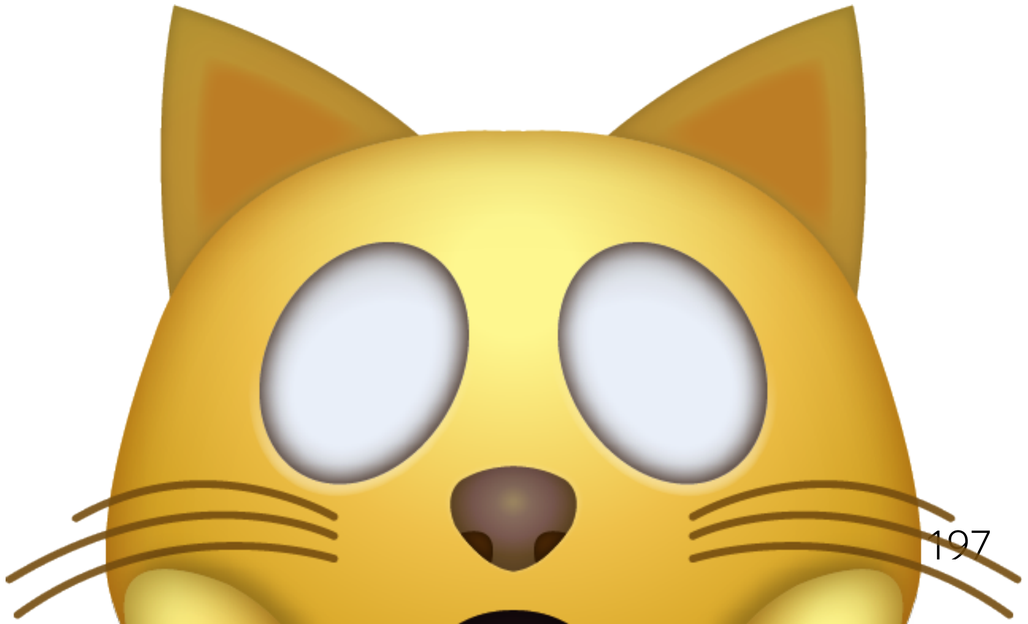
Структуры данных в TypeScript

- ~~Объединение (union)~~
- ~~Пересечение (intersection)~~
- ~~Перечислители (enum)~~
- Интерфейс (interface)
- Кортежи (tuple)



Структуры данных в TypeScript

- ~~Объединение (union)~~
- ~~Пересечение (intersection)~~
- ~~Перечислители (enum)~~
- Интерфейс (interface)
- ~~Кортежи (tuple)~~



Структуры данных в TypeScript

- ~~Объединение (union)~~
- ~~Пересечение (intersection)~~
- ~~Перечислители (enum)~~
- **Интерфейс (interface)**
- ~~Кортежи (tuple)~~

interface

Интерфейсы (interface)

Структура, схожая с классами, позволяющая описывать другие структуры, например классы и объекты, но при этом не имеющая их реализации.

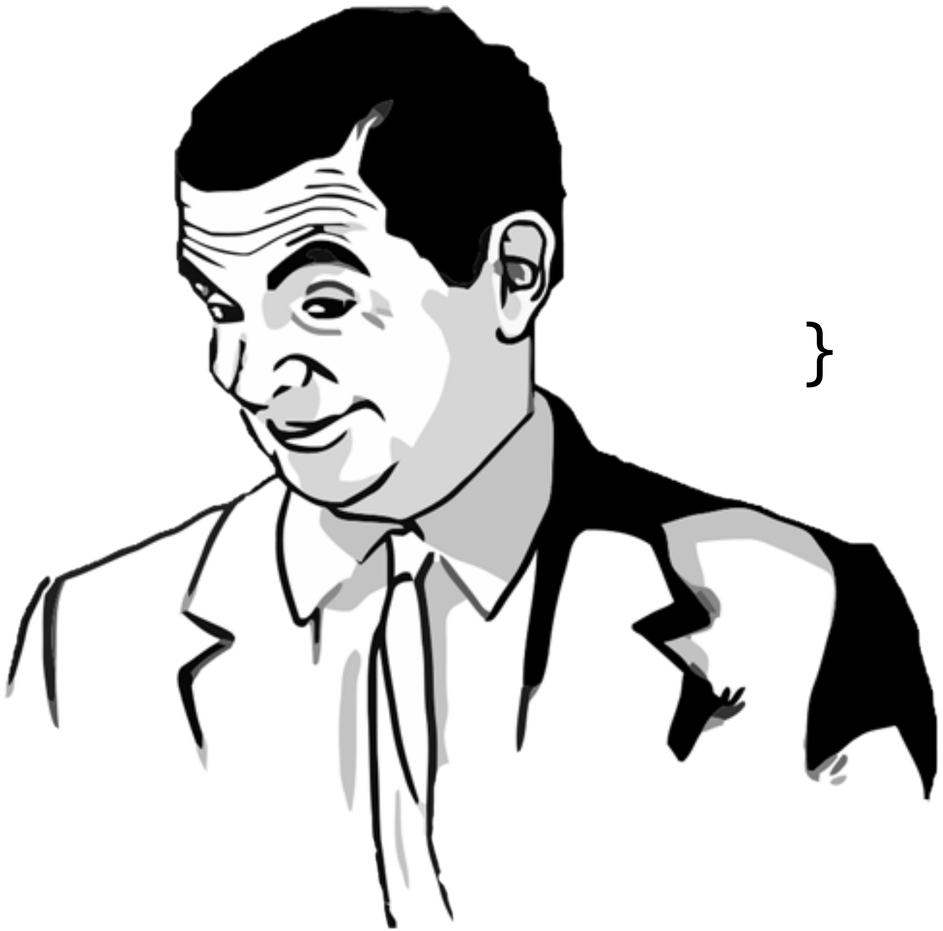
Да будет объект!

```
const obj = {
    sleep: true,
    type: 'red',
    age: 1
};
```

Какой тип имеет переменная «obj»?

Да будет объект!

```
const obj = {     // Type: ???
    sleep: true,
    type: 'red',
    age: 1
};
```



```
{
    sleep: boolean;
    type: string;
    age: number;
}
```

Да будет интерфейс!

```
const obj = {
    sleep: true,
    type: 'red',
    age: 1
};
```

```
interface IPanda {
    sleep: boolean,
    type: string,
    age: number
};
```

Да будет интерфейс!

```ts
interface IPanda {
    sleep: boolean;
    type: string;
    age: number;
}
```

```ts
const obj: IPanda = {
    sleep: true,
    type: 'red',
    age: 1
};
```

Да будет интерфейс!

```
interface IPanda {
    sleep: boolean;
    type: string;
    age: number;
}
```

```
const obj: IPanda = {
    sleep: true,
    type: 'red',
    age: 1
};
```

Да будет новый интерфейс!

```
interface IPanda {
    sleep: boolean;
    type: string;
    age: number;
}
```

```
interface IRedPanda {
    sleep: boolean;
    type: 'red';
    age: number;
}
```

Да будет новый интерфейс!

```
interface IPanda {
    sleep: boolean;
    type: string;
    age: number;
}
```

```
interface IRedPanda {
    sleep: boolean;
    type: 'red';
    age: number;
}
```

Да будет новый интерфейс!

```
interface IRedPanda {
    sleep: boolean;
    type: 'red';
    age: number;
}
```

Почему?

Rejected!

Зачем?

Одумайся!

Копипаста!

Не делай так!

Ты пьян!

Пожалуйста, расширь базовый интерфейс **IPanda**.

Да будет расширение!

```
interface IRedPanda extends IPanda {
    type: 'red';
}
```

Да будет расширение!

```
interface IRedPanda extends IPanda {
    type: 'red';
}
```

=

```
interface IRedPanda {
    sleep: boolean;
    type: 'red';
    age: number;
}
```

Важно!

```
interface IBase {
    age: number;
}


interface IFirst extends IBase {
    age?: number;  // Error
}
```

Фиксим проблемы

```
interface IBase {
    age: number;
}
```

```
interface IBase {
    age?: number;
}
```

```
interface ISecond extends IBase {
    age: number;
}
```

`interface`

Индексируемые типы

Индексируемые типы

```
const storage = {
    '/usr': true,
    '/fast-glob.json': true,
    '/bin': true,
    '/opt': true
}

interface IStorage {
    [prop: string]: boolean;
}
```

Индексируемые типы

```
interface IStorage {
    [prop: string]: boolean;
}

const storage: IStorage = {
    '/usr': true,
    '/fast-glob.json': true,
    '/bin': true,
    '/opt': true
}
```

Индексируемые типы

```
interface IStorage {
    [index: number]: string;
}

const obj: IStorage = {
    0: 'string'
};
```

Индексируемые типы

```
interface IStorage {
    [index: number]: string;
}

const obj: IStorage = {
    0: 'string'
};

obj.push(1);  // ???
```

Индексируемые типы

```
interface IStorage {
    [index: number]: string;
}

const obj: IStorage = {
    0: 'string'
};

obj.push(1);  // Property 'push' does not exist on type 'IStorage'
```

Индексируемые типы

```
interface IStorage {
    [index: number]: string;
}


const arr: IStorage = ['string'];


arr.push(1);  // Property 'push' does not exist on type 'IStorage'
```

Индексируемые типы

```ts
type IStorage = Record<number, string>;


const obj: IStorage = {
    0: 'string'
};
```

Вернуться
к содержанию

Обобщения

Обобщения (generics)

> Обобщённый тип (обобщение, дженерик) позволяет резервировать место для типа, который будет заменён на конкретный, переданный пользователем, при вызове функции или метода, а также при работе с классами.

https://canonium.com/articles/typescript-generics

Зачем?

Пример

```
function identity(arg: number): number {
    return arg;
}
```

Пример

```
function identity(arg: string): string {
    return arg;
}
```

Пример

```
function identity(arg: object): object {
    return arg;
}
```

Пример

```
function identity(arg: ololo): ololo {
    return arg;
}
```

Плохое решение

```
function identity(arg: any): any {
    return arg;
}
```

Плохое решение

```
function identity(arg: any): any {
    return arg;
}

const a = identity(1);  // Type: any
```

Плохое решение

```
function identity(arg: any): any {
    return arg;
}

const a = identity(1);  // ❌  Type: any
```

Почему плохое?

object

number

string

Хорошее решение

```
function identity<T>(arg: T): T {
    return arg;
}

 const number = identity(1);
```

Хорошее решение

```
function identity<T>(arg: T): T {
    return arg;
}

const number = identity(1);
```

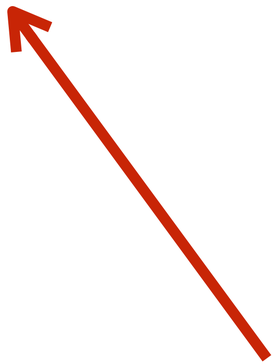
Хорошее решение

```
function identity<T>(arg: T): T {
    return arg;
}

const number = identity(1);
```

number

Хорошее решение

```
function identity<T>(arg: T): T {
    return arg;
}

const number = identity(1);
```



number

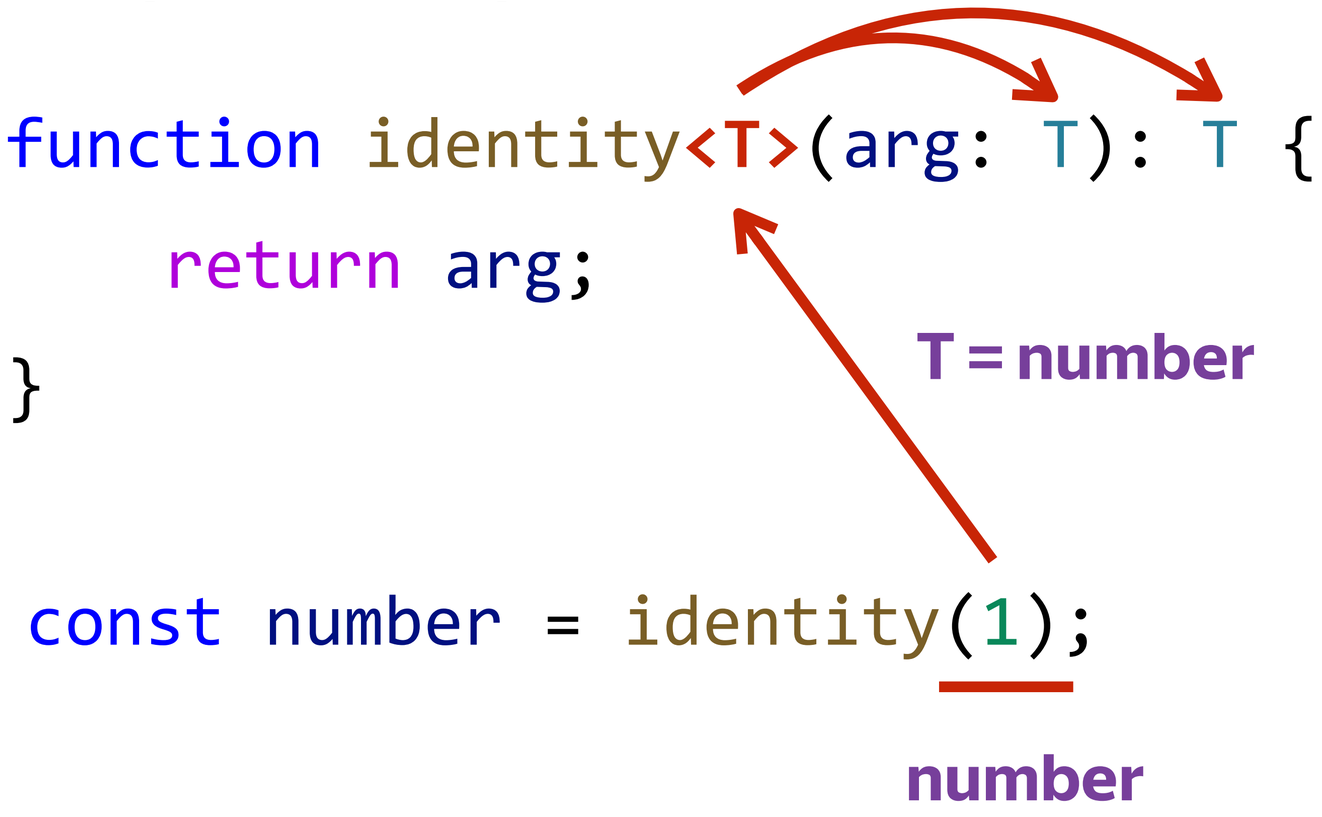
Хорошее решение

```
function identity<T>(arg: T): T {
    return arg;
}

const number = identity(1);
```

T = number

number

Хорошее решение



Хорошее решение

```
function identity<T>(arg: T): T {
    return arg;
}

const number = identity(1);  // Type: number
```

Так. Непонятно. Зачем?

Зачем нужны обобщения?

- Универсальность кодовой базы
- Возможность ограничения типов, с которыми может работать функция

Простой пример

Простой пример

```
interface IItem {
    name: string;
}

const storage = new Map<string, IItem>();
```

Простой пример

```
interface IItem {
    name: string;
}

const storage = new Map<string, IItem>();
```

Простой пример

```
interface IItem {
    name: string;
}

const storage = new Map<string, IItem>();
```

Простой пример

```
interface IItem {
    name: string;
}

const storage = new Map<string, IItem>();

storage.set('key', { name: 'value' });  // 👍
```

Простой пример

```
interface IItem {
    name: string;
}

const storage = new Map<string, IItem>();


storage.set('key', { name: 'value' });  // 👍

storage.set('key', 'value');  // ❌ Compilation error
```

Ограничение обобщений

А есть ли тут ошибка?

```
function getLength<T>(arg: T): number {
    return arg.length;
}
```

Ошибка здесь!

```
function getLength<T>(arg: T): number {
    return arg.length;
}
```

А почему?

```
function getLength<T>(arg: T): number {
    return arg.length;
}
```

А почему?

T = неизвестный тип

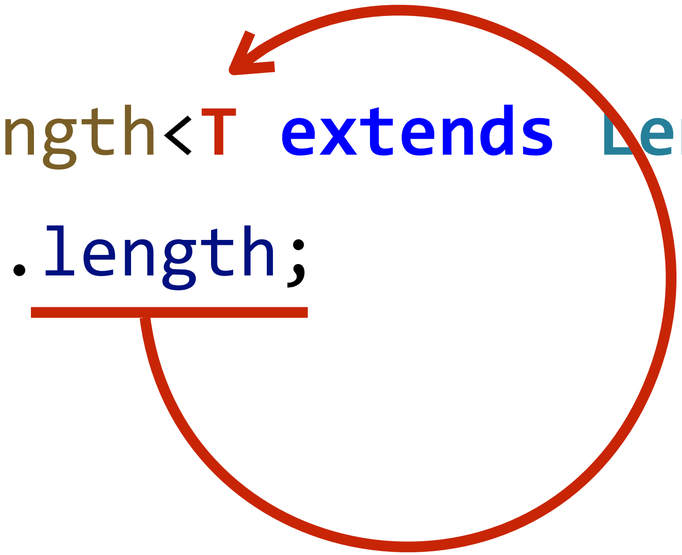
```
function getLength<T>(arg: T): number {
    return arg.length;
}
```

Как починить?

```
interface Lengthwise {
    length: number;
}

function getLength<T extends Lengthwise>(arg: T): number {
    return arg.length;
}
```

Как починить?

```
interface Lengthwise {
    length: number;
}

function getLength<T extends Lengthwise>(arg: T): number {
    return arg.length;
}
```

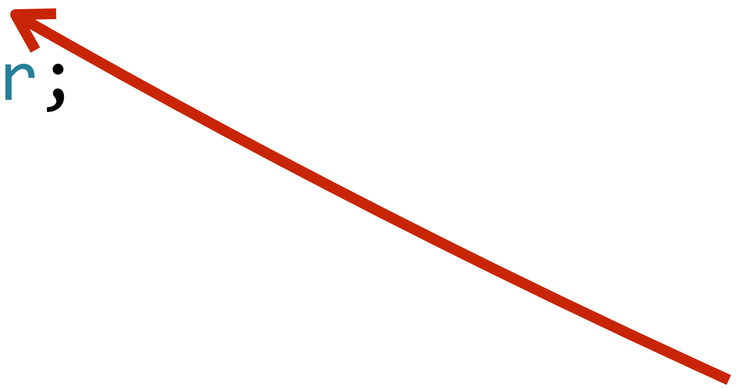


Как починить?

```
interface Lengthwise {
    length: number;
}

function getLength<T extends Lengthwise>(arg: T): number {
    return arg.length;
}
```

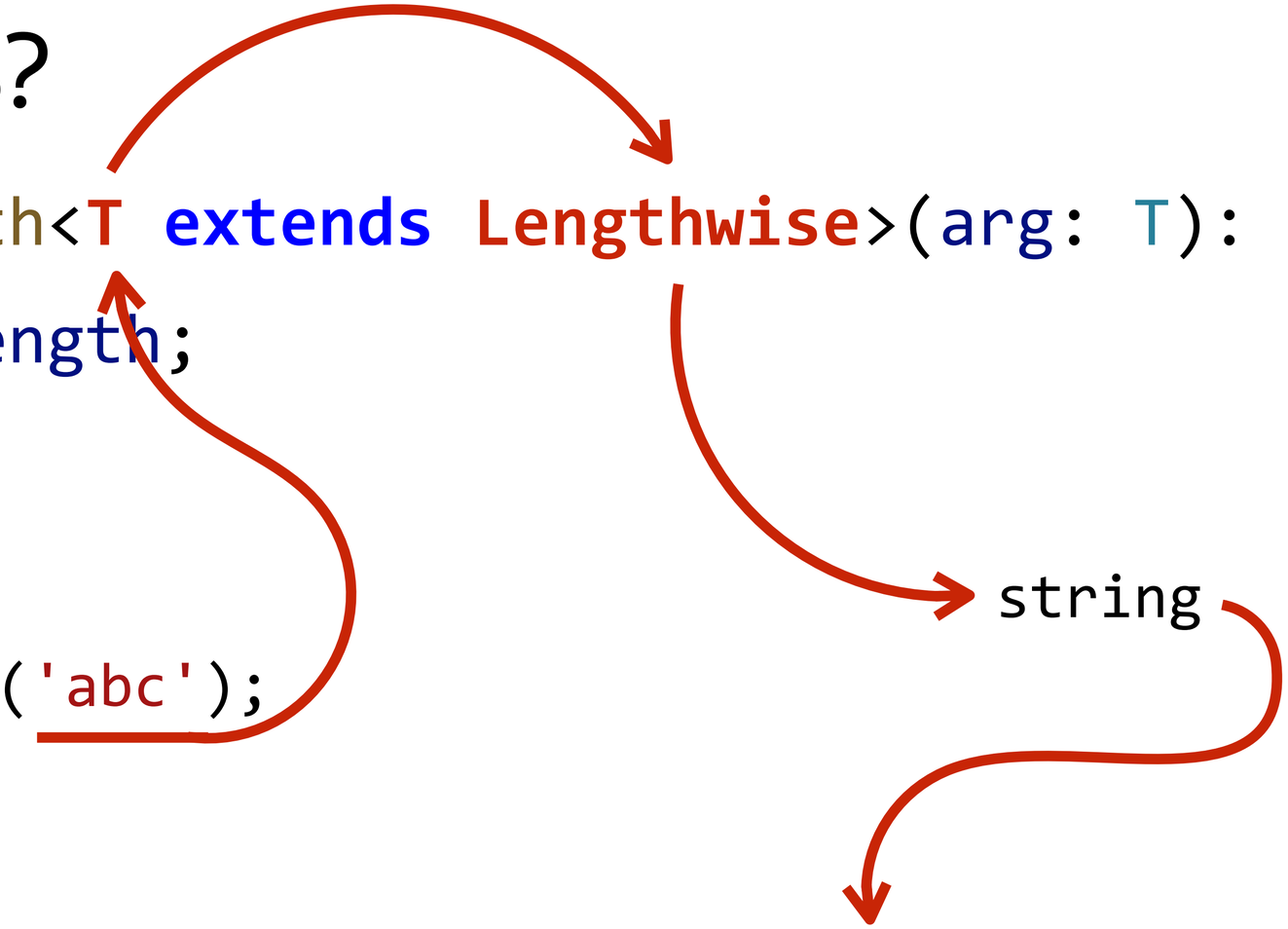
Как починить?

```
interface Lengthwise {
    length: number;
}

function getLength<T extends Lengthwise>(arg: T): number {
    return arg.length;
}
```



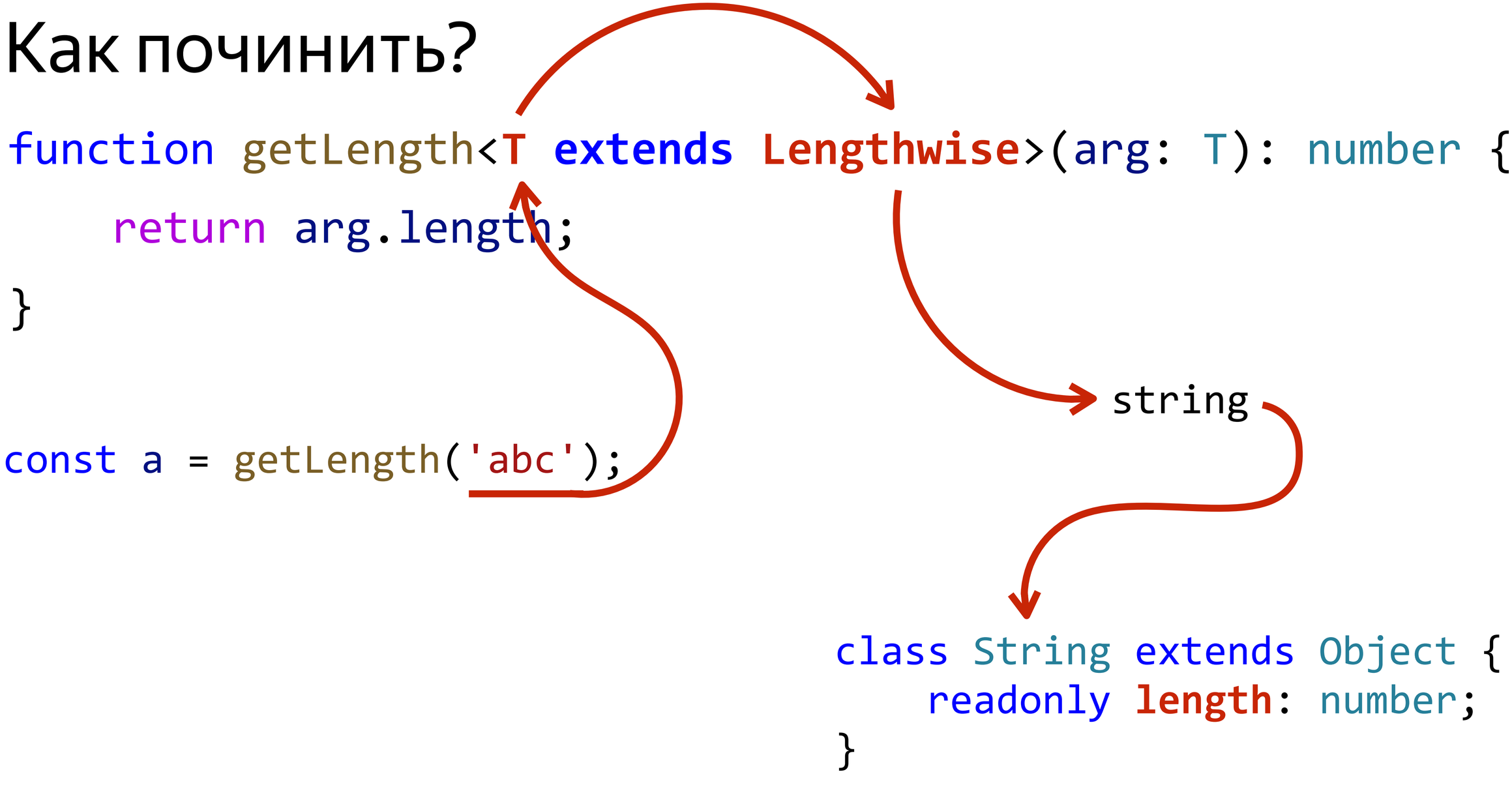
Как починить?

```
function getLength<T extends Lengthwise>(arg: T): number {
    return arg.length;
}

const a = getLength('abc');
```




```
class String extends Object {
    // ...
}
```

Как починить?



Как починить?

```
function getLength<T extends ArrayLike<any>>(arg: T): number {
    return arg.length;
}
```

Реальный пример

fast-glob

https://github.com/mrmlnc/fast-glob

Реальный пример

```
function getWorks<T>(
    source: Pattern | Pattern[],
    _Reader: new (options: IOptions) => Reader,
    opts?: IPartialOptions
): T[] {}
```

Реальный пример

```
function getWorks<T>(
    source: Pattern | Pattern[],
    _Reader: new (options: IOptions) => Reader,
    opts?: IPartialOptions
): T[] {}
```

Реальный пример

```
function getWorks<T>(
    source: Pattern | Pattern[],
    _Reader: new (options: IOptions) => Reader,
    opts?: IPartialOptions
): T[] {}
```

```
const works = getWorks<EntryItem[]>(source, ReaderSync, opts);
```

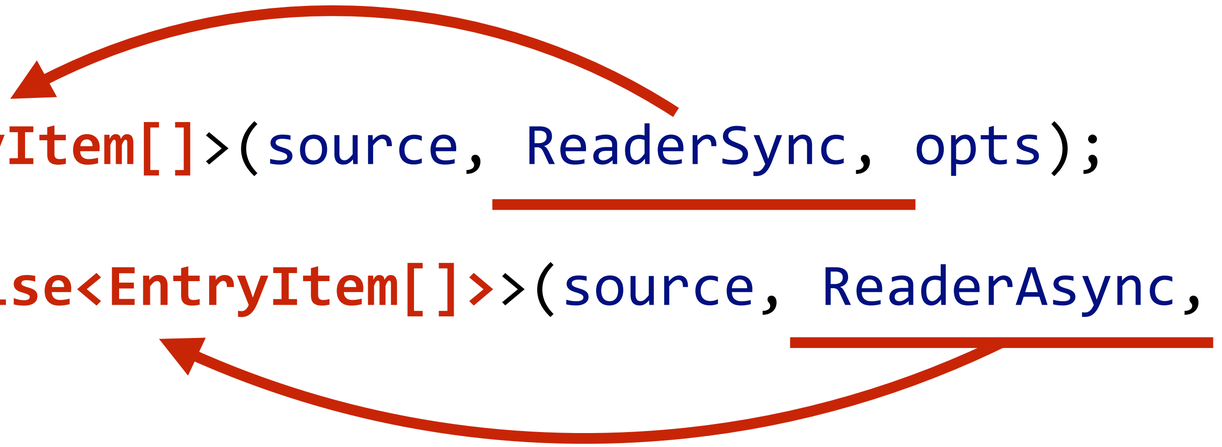
```
const works = getWorks<Promise<EntryItem[]>>(source, ReaderAsync, opts);
```

Реальный пример

```
function getWorks<T>(
    source: Pattern | Pattern[],
    _Reader: new (options: IOptions) => Reader,
    opts?: IPartialOptions
): T[] {}
```

```
const works = getWorks<EntryItem[]>(source, ReaderSync, opts);
```

```
const works = getWorks<Promise<EntryItem[]>>(source, ReaderAsync, opts);
```



Вернуться
к содержанию

Когда нужен TS/Flow/…?

Когда нужен TypeScript/Flow/Reason/…?

- Разрабатываете проект командой (не двумя людьми free time)
- Разрабатываете проекты (не маленькие)
- Планируете поддерживать и развивать свой проект в будущем
- Много работаете со структурами данных (объекты, например)
- Хотите получать от кода максимум (автокомплит, проверки, …)
- Хотите анализировать написанный код с помощью утилит

Happy coding!

Денис Малиночкин

Разработчик инфраструктуры
разработки интерфейсов