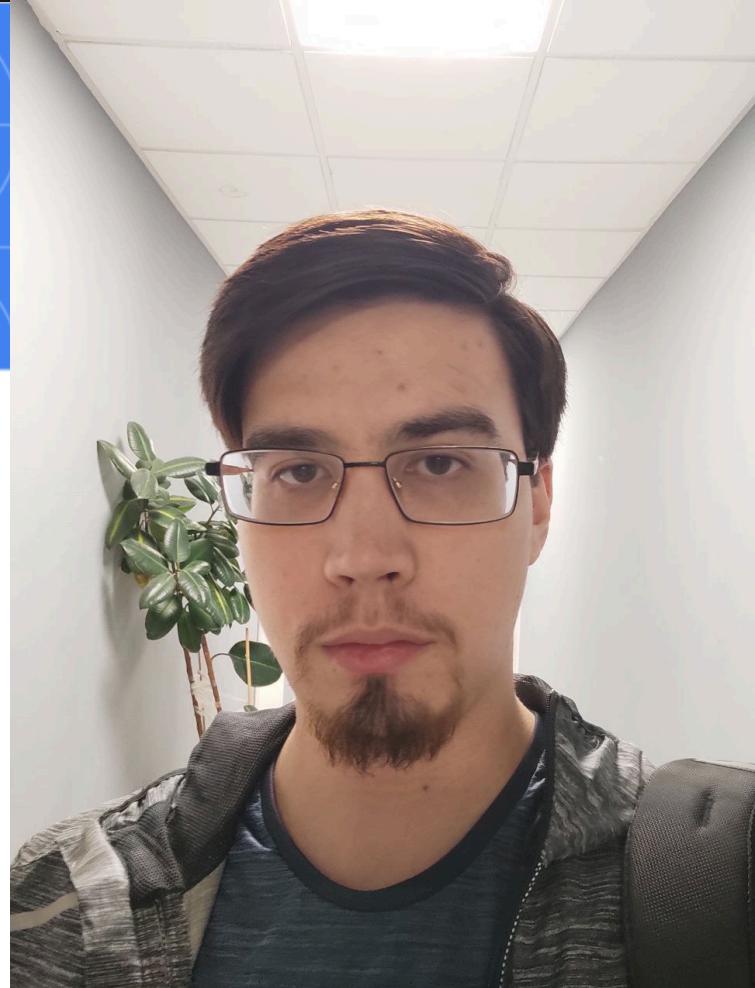


devfest  
OMSK



# Безопасный TypeScript

Денис Чернов  
SM Lab, ведущий разработчик





# Денис Чернов

↗ @zede\_code

¶ @izede

GitHub icon @Sdju



# Зачем нам **TypeScript**?

## 1. Обнаруживаем ошибки раньше

- Видим ошибки не запуская код
- Проверяем соответствия на CI/CD
- Видим несоответствия во время написания кода

# Зачем нам **TypeScript**?

## 2. Устанавливаем контракты

Контракт - договоренности между отдельными участниками

- Согласованность между Frontend и Backend-ом
- Корректная работа между библиотеками и приложениями
- Единое представление в рамках одной кодовой базы

# Зачем нам **TypeScript**?

## 3. Упрощаем переиспользование кода

- Нам не нужно гадать что и как используется
- Четко фиксируем что ожидаем от конкретного участка кода
- Облегчаем понимание другим разработчиком

## Зачем нам **TypeScript**?

### 4. Служит своеобразной документацией

- Даём описание того что отдаём и что принимаем
- Вынуждаем разработчиков поддерживать её в актуальном состоянии

## Зачем нам **TypeScript**?

### 5. Улучшает подсказки в редакторе

- Теперь IDE четко понимает что нам предлагать
- IDE легко показывает ошибки прямо в процессе работы

# Особенности типизации TS

# Явная

- Явное определение типов

```
let name = 'John Doe'
```

# Строгая

**Строгая типизация** - не позволяет смешивать типы данных в операциях

Не путать со статической типизацией

```
const weight = 70
const person = { name: 'John Doe', weight: 60 }

console.log(weight + person.weight);
// Output: 130
```

# Утиная

"Если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка."

**Утиная типизация** - это концепция, при которой тип объекта определяется его свойствами и методами, а не явным объявлением типа.

# Утиная

```
type Duck = { swim(): void, quack(): void }
function duckTest(a: any): a is Duck {
    if (!('quack' in a) || (typeof a.quack === 'function')) {
        return false
    }
    if (!('swim' in a) || (typeof a.swim === 'function')) {
        return false
    }
    return true
}

function action(a: unknown) {
    if (duckTest(a)) {
        a.swim()
    }
}
```

# Структурная

```
type Duck = { eat(): void }
type Cat = { eat(): void }

type DuckIsCat = Duck extends Cat ? true : false; // true
type CatIsDuck = Cat extends Duck ? true : false; // true

let duck: Duck = { eat() }
let cat: Cat = duck // OK!
```

```
type Duck = { eat(): void }
type Cat = { eat(): void }

type DuckIsCat = Duck extends Cat ? true : false; // true
type CatIsDuck = Cat extends Duck ? true : false; // true

let duck: Duck = { eat() }
let cat: Cat = duck // OK!
```

# Проблема

- С точки зрения TS сущности совместимы
- С точки зрения JS сущности не совместимы

# Причина

- TS использует **структурную типизацию**, а не **номинальную**

# Номинальная типизация

- Типизация основанная на связи идентификаторов
- Не важно насколько похожи структуры, если связи между ними нет, то типы не совместимы
- Увеличивает предсказуемость
- Ближе по логике работы с JS

```
class Duck {
  eat() {
    console.log('duck is eating')
  }
}
class Cat {
  eat() {
    console.log('cat is eating')
  }
}

type DuckIsCat = Duck extends Cat ? true : false; // true
type CatIsDuck = Cat extends Duck ? true : false; // true
```

# Что мы сделали?

- Добавили номинальную типизацию засчет брандирования
- Кошки больше не могут быть утками
- Решает проблему только для классов / объектов
- Не самый удобный в использовании
- Требует много манипуляций
- Никак не влияет на рантайм

# Улучшаем решение

```
function getOffset(size: number) {  
    return size * 0.2 + 10  
}  
  
let widthInPercents = 10  
getOffset(widthInPercents) // OK!
```

# Брендирование

- Работает с примитивами
- Не создает нагрузки в рантайме
- Требует избыточного кода
- Бренды отклеиваются после операторов
- Брендированные значения не могут быть ключами объектов

# Решения от самого TypeScript

- Есть предложения на добавление ключевого слова `nominal`
- Есть предложения на добавление ключевого слова `instanceof` для типов
- Есть предложения на возможность быть ключами у брандированных строк
- Уже существует тип, который работает по номинальной типизации...

# Перечисления

# Enums

- Единственный тип с номинальной типизацией
- Единственный TS тип непосредственно влияющий на JS
- Есть сценарии с неожиданным поведением

```
enum Roles {  
    Admin,  
    Reader  
}
```

# Работа с Enum

- Всегда инициализировать варианты Enum
- Использовать ESLint **prefer-enum-initializers**
- Избегать **const enum**
- Предпочитать строковые значения численным

# Альтернатива Enum

- Использовать **union type**
- Использовать объекты с **as const**

# Используем **as const**

```
enum Roles {  
    Admin = 'Admin',  
    Reader = 'Reader'  
}
```

Работаем с объектами

# В чём разница?

```
interface Obj {  
    version1(param: string): void  
  
    version2: (param: string) => void  
}
```

# Собака гавкующая на щенков

```
interface Dog {
  barkAt(dog: Dog): void
}

interface SmallDog extends Dog {
  whimper(): void
}

const brian: Dog = {
  barkAt(smallDog: SmallDog) {
    smallDog.whimper()
  },
};

const normalDog: Dog = {
  barkAt() {},
}

brian.barkAt(normalDog)
```

## Почему это происходит

- Раньше функции тоже могли принимать более широкие типы
- Теперь при использовании **strict** это не пропускается
- Методам оставили поддержку более широкого типа
- Иначе все сломается

**interface**  $\neq$  **type**

# Очевидное

- **interface** не может описать примитивы
- **type** позволяет описывать более сложные типы
- Оба могут описывать объекты и их расширения

# **interface** может быть расширен

```
interface Animal {  
    name: string  
}  
  
interface Animal {  
    weight: number  
}  
  
// Animal = { name: string, weight: number }
```

# Индексирование по-умолчанию

```
type Animal = {  
    tag: 'animal',  
    name: 'some animal'  
}  
  
declare var animal: Animal;  
  
const handleRecord = (obj:Record<string, unknown>) => {}  
  
const result = handleRecord(animal)
```

satisfies

## Предпочитайте использование **satisfies**

- Не дает случайно обмануть компилятор как **as**
- Значение важнее чем указанный тип

```
let a = "123"
const b = "123"
let c: number | string = '123'
let d = 123 as number | string
let e = 123 satisfies number | string
let f = false satisfies number | string
```

# Определение типа

```
type Palette = 'red' | 'green' | 'blue'

const palette = {
  red: '#FF0000',
  green: '#00FF00',
  blue: '#0000FF',
} as Record<Palette, string>
// OK!
```

# Улучшаем работу TypeScript

# Как влиять на поведение TypeScript?

- Флаги и настройки в `tsconfig.json`
- Переопределение интерфейсов методов
- Подбирая более удачные синтаксисы в нужные моменты времени

# Какие флаги включить?

- **strict** - Базовая опция включающая большую часть строгих проверок
- **exactOptionalPropertyTypes** - в случае `?:` не дает значениям быть `undefined`
- **noFallthroughCasesInSwitch** - не дает забыть вам поставить `break` в конце `case`
- **noUncheckedIndexedAccess** - не доверяет индексации без проверки

# TS Reset

Improved TypeScript's  
Built-In Typings



Matt Pocock



# ts-reset

- Подключается 1 строкой
- `fetch` / `JSON.parse` возвращают `unknown`, а не `any`
- Поиск в массивах больше не ограничен типом элементов массива
- `.filter(Boolean)` убирает из типов элемента `falsy-значения`

```
import "@total-typescript/ts-reset";
```

# TypeScript и рантайм

# TypeScript и рантайм

1. У нас есть возможности JS!
2. Делаем описание типа используя JS
3. Выгружаем из описания TS-схему
4. Валидируем значения при необходимости в рантайме

# Валидаторы

- Проверяем значение на соответствие правилам
- Простойка между TS и JS
- Устраняет необходимость писать валидацию вручную
- Проверки соответствия правилам и в рантайме и в типах

# Существующие решения

- **zod** - популярный и мощный инструмент
- **joi** - крайне мощный инструмент
- **yup** - популярный и мощный инструмент
- **class-validator** - основан на декораторах
- **valibot** - супер-легковесный инструмент

# Пример с **zod**

```
import { z } from 'zod'

const A = z.string()
type A = z.infer<typeof A> // string

const a1: A = 12 // ERROR!
const a2: A = "asdf" // compiles
```

# Когда использовать валидаторы?

- Когда TS не может гарантировать правильность данных
- Когда TS не может покрыть все возможные правила
- Когда валидацию нужно проводить в рантайме

Итоги

**TypeScript** - мощный инструмент, но им нужно управлять

Похожие решения могут быть разными на деле

TypeScript не всемогущий

Типизация хорошо, но не слишком увлекайтесь

# Задавайте вопросы

-  @zede1697
-  @izede
-  @izede



