TypeScript

Часть II

Старков Дима

Сегодня

- Вывод типов

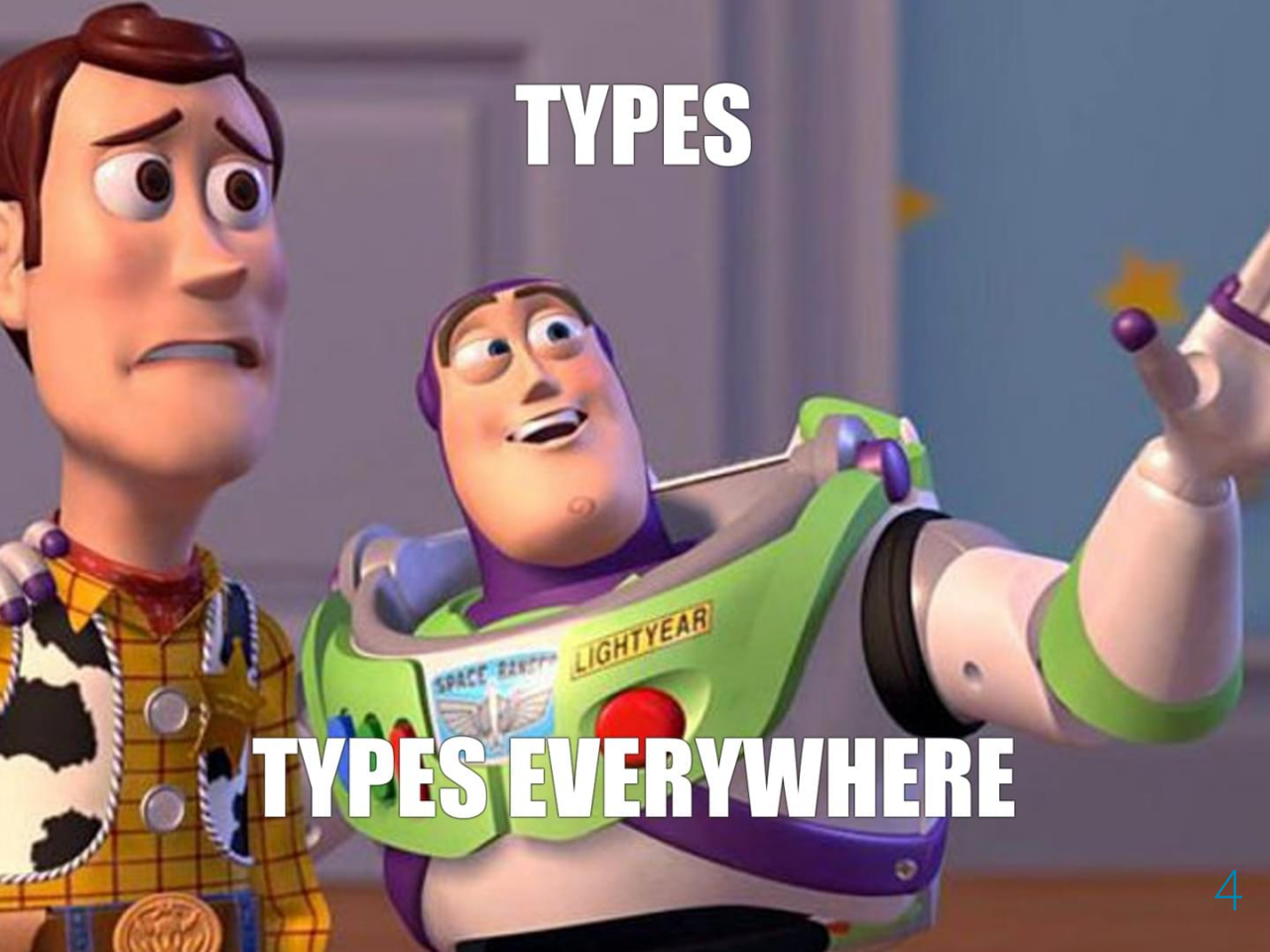
- Структурная типизация
- Более сложные типы
- Обобщенные типы
- Type Guards

TypeScript?

- Спасет от выстрелов себе в ногу

- ESNext прямо сейчас
- Средство против TypeError
- Пишет код за вас
- Документация к коду

Но...


TYPES
LIGHTYEAR
TYPES EVERYWHERE

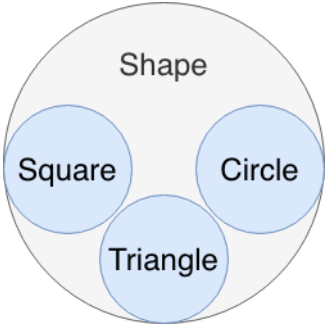
Вывод типов

```
let n: number = 42
let s: string = 'Hello, world!'
let a: number[] = [1, 2, 3, 4]

let n = 42
let s = 'Hello, world!'
let a = [1, 2, 3, 4]
```

Наиболее общий тип

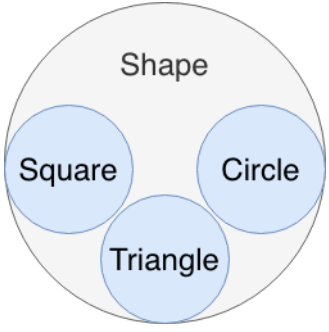
```
let shapes = [new Circle(), new Square()]
```

```
shapes.push(new Triangle())
```



Наиболее общий тип

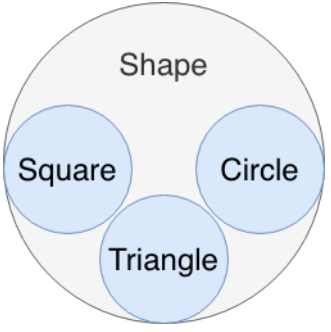
```
let shapes = [new Circle(), new Square()]

// Argument of type 'Triangle'
// is not assignable to parameter of type 'Square | Circle'.
shapes.push(new Triangle())
```



Наиболее общий тип

```
let shapes = [new Circle(), new Square()]
```

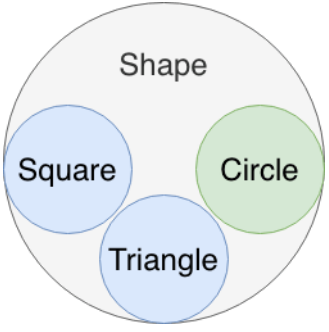
```
shapes.push(new Triangle())
```



Наиболее общий тип

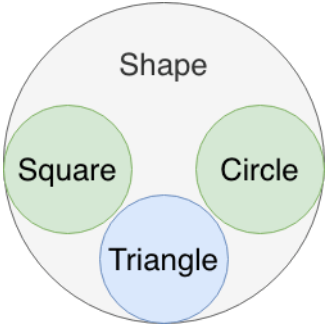
```
let shapes = [new Circle(), new Square()]
```

```
shapes.push(new Triangle())
```



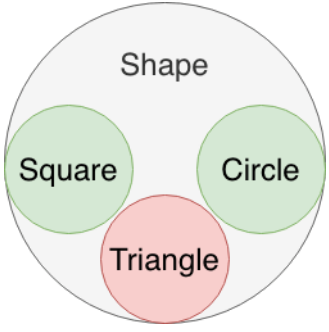
Наиболее общий тип

```
let shapes = [new Circle(), new Square()]


shapes.push(new Triangle())
```



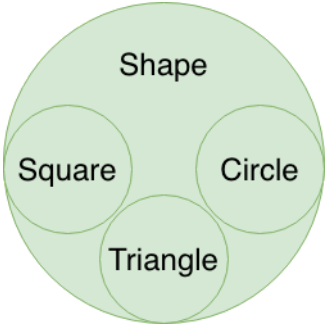
Наиболее общий тип

```
let shapes = [new Circle(), new Square()]

// Argument of type 'Triangle'
// is not assignable to parameter of type 'Square | Circle'.
shapes.push(new Triangle())
```



Наиболее общий тип

```
let shapes: Shape[] = [new Circle(), new Square()]


shapes.push(new Triangle())
```



Совместимость типов

```
class Human {
    name: string
}

class Robot {
    name: string
}

let human: Human = new Robot()
```

Проверка типа в runtime

```
function addShape(shapes: Shape[], obj: object) {
    if (obj instanceof Shape) {
        shapes.push(obj as Shape)
    }

    throw new TypeError('Argument is not instanceof Shape')
}
```

Проверка типа в runtime

```
function addShape(shapes: Shape[], obj: object) {
    if (obj instanceof Shape) {
        shapes.push(obj as Shape)
    }

    throw new TypeError('Argument is not instanceof Shape')
}
```

Проверка типа в runtime

```
function addShape(shapes: Shape[], obj: object) {
    if (obj instanceof Shape) {
        shapes.push(obj as Shape)
    }

    throw new TypeError('Argument is not instanceof Shape')
}
```

Проверка типа в runtime

```
function addShape(shapes: Shape[], obj: object) {
    if (obj instanceof Shape) {
        shapes.push(obj as Shape)
    }

    throw new TypeError('Argument is not instanceof Shape')
}
```

Проверка типа в runtime

```
function addShape(shapes: Shape[], obj: object) {
    if (obj instanceof Shape) {
        shapes.push(obj as Shape)
    }

    throw new TypeError('Argument is not instanceof Shape')
}
```

Type Guard

```
function addShape(shapes: Shape[], obj: object) {
    if (obj instanceof Shape) {
        shapes.push(obj)
    }

    throw new TypeError('Argument is not instanceof Shape')
}
```

TypeScript крут.
Но можем ли мы описать весь JavaScript?

Вспомним TypeScript 1.0

- Интерфейсы

- Классы
- Обобщенные типы
- Перегрузки функций

Чего еще желать?

```
// String.split
split(separator: ?, limit: number): string[]
```

```
// String.split
split(separator: string | RegExp, limit: number): string[]
```

```
// String.split
split(separator: string | RegExp, limit: number): string[]
```

Решение: Union Types

Union Type Guard

```
function negate(n: string | number) {
    if (typeof n === 'string') {
        return '-'.concat(n);
    } else {
        return -n;
    }
}
```

Union Type Guard

```
function negate(n: string | number) {
    if (typeof n === 'string') {
        return '-'.concat(n);
    } else {
        return -n;
    }
}
```

Union Type Guard

```
function negate(n: string | number) {
    if (typeof n === 'string') {
        return '-'.concat(n);
    } else {
        return -n;
    }
}
```

Union Type Guard

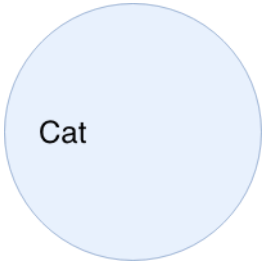
```
function negate(n: string | number) {
    if (typeof n === 'string') {
        return '-'.concat(n);
    } else {
        return -n;
    }
}
```

Union Type Guard

```
function negate(n: string | number) {
    if (typeof n === 'string') {
        return '-'.concat(n);
    }

    return -n;
}
```

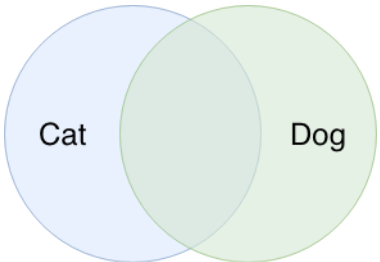
Intersection Types

```
type Cat = {
    purr()
}
```



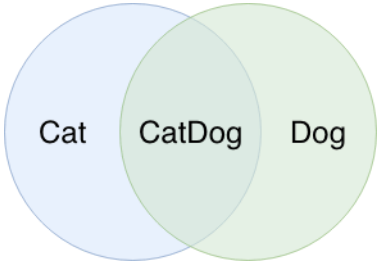
Intersection Types

```
type Cat = {
    purr()
}

type Dog = {
    woof()
}
```



Intersection Types

```
type Cat = {
    purr()
}

type Dog = {
    woof()
}

type CatDog = Cat & Dog
```



Type Alias

```
// String.split
split(separator: string | RegExp, limit: number): string[]
```

Type Alias

```
type StringOrRegExp = string | RegExp

// String.split
split(separator: StringOrRegExp, limit: number): string[]
```

Type vs Interface

```
type Point = {
    x: number
    y: number
}
```

```
interface Point {
    x: number
    y: number
}
```

- implements **interface**
- **Type1 | Type2**

Тип ≡ Множество

- Можем объединять типы |

- Можем пересекать типы &
- Можем вычитать из одного типа другой

Фух, теперь точно всё...

А вот и нет!

```
function get(obj, keyName) {

    return obj[keyName]
}


function get(obj: any, keyName: string): any {
    return obj[keyName]
}

// TypeError: Cannot read property 'prototype' of null
get(null, 'prototype')
```

А вот и нет!

```
function get(obj, keyName) {

    return obj[keyName]
}


function get(obj: any, keyName: string): any {
    return obj[keyName]
}

// TypeError: Cannot read property 'prototype' of null
get(null, 'prototype')
```

А вот и нет!

```
function get(obj, keyName) {

    return obj[keyName]
}


function get(obj: any, keyName: string): any {
    return obj[keyName]
}

// TypeError: Cannot read property 'prototype' of null
get(null, 'prototype')
```

А вот и нет!

```
function get(obj, keyName) {

    return obj[keyName]
}


function get(obj: any, keyName: string): any {
    return obj[keyName]
}

// TypeError: Cannot read property 'prototype' of null
get(null, 'prototype')
```

Нужно уметь обрабатывать значения разных типов идентичным образом

Кажется нам нужен...


any
Это полиморфизм?

Обобщенные типы

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

Обобщенные типы

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

Обобщенные функции

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

identity('string') // T is string
identity(12131415) // T is number
identity([4, 8, 15, 16, 23, 42]) // T is number[]
```

Встроенные обобщенные типы

```
const fib: Array<number> = [1, 1, 2, 3, 5]

// Argument of type 'string'
// is not assignable to parameter of type 'number'.
fib.push('1')

const map: Map<number, string> = new Map()

// Argument of type 'number'
// is not assignable to parameter of type 'string'.
map.set(1, 1)
```

Обобщенные интерфейсы

```
interface IStack<TItem> {
    push(item: TItem)
    pop(): TItem
}

let numStack: IStack<number> = [1, 2, 3]
```

Обобщенные интерфейсы

```
interface IStack<number> {
    push(item: number)
    pop(): number
}

let numStack: IStack<number> = [1, 2, 3]
```

Обобщенные типы

```
type AsyncResult<TResult> = Promise<TResult> | TResult

let result: AsyncResult<string> = Promise.resolve('200')
let result: AsyncResult<string> = '200'
```

Обобщенные типы

```
type AsyncResult<string> = Promise<string> | string

let result: AsyncResult<string> = Promise.resolve('200')
let result: AsyncResult<string> = '200'
```

Обобщенные классы

```
class Stack<TItem> implements IStack<TItem> {
    private state: TItem[]

    constructor() {
        this.state = []
    }

    push(item: TItem) {
        this.state.push(item)
    }

    pop(): TItem {
        return this.state.pop()
    }
```

Обобщенные классы

```
class Stack<TItem> implements IStack<TItem> {
    private state: TItem[] = []

    push(item: TItem) {
        this.state.push(item)
    }

    pop(): TItem {
        return this.state.pop()
    }
}
```

Обобщенные типы

```
interface ISwim {
    swim()
}

class Dog implements ISwim {
    swim() { ... }
}

class Duck implements ISwim {
    swim() { ... }
}
```

Ограничения на обобщенные типы

```
function swimTogether<
    T1 implements ISwim,
    T2 implements ISwim
>(firstPal: T1, secondPal: T2) {
    firstPal.swim()
    secondPal.swim()
}
```

Обобщенные типы

```
type TypeName<T> =
    T extends string ? 'string' :
    T extends number ? 'number' :
    T extends boolean ? 'boolean' :
    T extends undefined ? 'undefined' :
    T extends Function ? 'function' :
    'object'
```

Обобщенные типы

```
type TypeName<string> =
    string extends string ? 'string' :
    T extends number ? 'number' :
    T extends boolean ? 'boolean' :
    T extends undefined ? 'undefined' :
    T extends Function ? 'function' :
    'object'
```

Обобщенные типы

```
type TypeName<number> =
    number extends string ? 'string' :
    number extends number ? 'number' :
    T extends boolean ? 'boolean' :
    T extends undefined ? 'undefined' :
    T extends Function ? 'function' :
    'object'
```

Наша функция

```
function get(obj: any, keyName: string): any {
    return obj[keyName]
}
```

Наша функция

```
function get<T>(obj: T, keyName: string): any {
    return obj[keyName]
}
```

Хотим знать список полей объекта
и их типы на этапе компиляции

Решение: Lookup Types и keyof

Lookup типы

```
interface IUser {
    login: string
    age: number
    gender: 'male' | 'female'
}

let login: IUser['login']
let login: string

let loginOrAge: IUser['login' | 'age']
let loginOrAge: string | number
```

keyof

```
interface IUser {
    login: string
    age: number
    gender: 'male' | 'female'
}

let key: keyof IUser
let key: 'login' | 'age' | 'gender'
```

Наша простая функция

```
function get(obj, keyName) {
    return obj[keyName]
}
```

Наша ~~простая~~ функция

```
function get<T>(obj: T, keyName: keyof T): T[keyof T] {
    return obj[keyName]
}
```

Наша ~~простая~~ функция

```
function get<T>(obj: T, keyName: keyof T): T[keyof T] {
    return obj[keyName]
}

let a: number = get({ a: 1 }, 'a')
```

Наша ~~простая~~ функция

```
function get<{ a: 1 }>(obj: T, keyName: keyof T): T[keyof T] {
    return obj[keyName]
}

let a: number = get({ a: 1 }, 'a')
```

Наша ~~простая~~ функция

```
function get<{ a: 1 }>(obj: T, keyName: 'a'): T['a'] {
    return obj[keyName]
}

let a: number = get({ a: 1 }, 'a')
```

Наша ~~простая~~ функция

```
function get<{ a: 1 }>(obj: T, keyName: 'a'): number {
    return obj[keyName]
}

let a: number = get({ a: 1 }, 'a')
```

Наша ~~простая~~ функция

```
function get<T>(obj: T, keyName: keyof T): T[keyof T] {
    return obj[keyName]
}

let a: number = get({ a: 1 }, 'a')

// Argument of type '"c"'
// is not assignable to parameter of type '"a" | "b"'.
let c: undefined = get({ a: 1, b: 2 }, 'c')
```

Наша ~~простая~~ функция

```
function get<T, K extends keyof T>(obj: T, keyName: K): T[K] {
    return obj[keyName]
}

let a: number = get({ a: 1 }, 'a')



let c: undefined = get({ a: 1, b: 2 }, 'c')
```

Перерыв

А что там в es5?

```
interface IUser {
    login: string
    age: number
    gender: 'male' | 'female'
}

const user = { login: 'dimastark', age: 21, gender: 'male' }
const readonlyUser: ? = Object.freeze(user)
```

А что там в es5?

```
interface IFrozenUser {
    readonly login: string
    readonly age: number
    readonly gender: 'male' | 'female'
}

const user = { login: 'dimastark', age: 21, gender: 'male' }
const readonlyUser: IFrozenUser = Object.freeze(user)
```

Решение: Mapped Types

Mapped Types

```
interface IUser {
    login: string
    age: number
    gender: 'male' | 'female'
}

type Readonly<T> = {
    readonly [P in 'login' | 'age' | 'gender']: T[P];
};

const user = { login: 'dimastark', age: 21, gender: 'male' }
const readonlyUser: Readonly<IUser> = Object.freeze(user)
```

Mapped Types + keyof

```
interface IUser {
    login: string
    age: number
    gender: 'male' | 'female'
}

type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};

const user = { login: 'dimastark', age: 21, gender: 'male' }
const readonlyUser: Readonly<IUser> = Object.freeze(user)
```

infer

```
type ValueOf<T> = T extends {
    [key: string]: infer U
} ? U : never;

ValueOf<{ a: string, b: string }>  // string
ValueOf<{ a: string, b: number }>  // string | number
```

Mapped Types

```
interface IUser {
    login: string
    birthDate: {
        year: number
        month: number
        day: number
    }
    gender: 'male' | 'female'
}
```

Mapped Types

```
type DeepReadonly<T> = {
    [P in keyof T]:
        T[P] extends (infer U)[] ? DeepReadonly<U>[] :
        T[P] extends object ? DeepReadonly<T[P]> :
        readonly T[P];
};
```

Mapped Types

```
type DeepReadonly<T> = {
    [P in keyof T]:
        T[P] extends (infer U)[] ? DeepReadonly<U>[] :
        T[P] extends object ? DeepReadonly<T[P]> :
        readonly T[P];
};
```

Mapped Types

```
type DeepReadonly<T> = {
    [P in keyof T]:
        T[P] extends (infer U)[] ? DeepReadonly<U>[] :
        T[P] extends object ? DeepReadonly<T[P]> :
        readonly T[P];
};
```

Mapped Types

```
type DeepReadonly<T> = {
    [P in keyof T]:
        T[P] extends (infer U)[] ? DeepReadonly<U>[] :
        T[P] extends object ? DeepReadonly<T[P]> :
        readonly T[P];
};
```

Mapped Types

```
type DeepReadonly<T> = {
    [P in keyof T]:
        T[P] extends (infer U)[] ? DeepReadonly<U>[] :
        T[P] extends object ? DeepReadonly<T[P]> :
        readonly T[P];
};
```

Mapped Types

```
type DeepReadonly<T> = {
    [P in keyof T]:
        T[P] extends (infer U)[] ? DeepReadonly<U>[] :
        T[P] extends object ? DeepReadonly<T[P]> :
        readonly T[P];
};
```

Mapped Types

```
type DeepReadonly<T> = {
    [P in keyof T]:
        T[P] extends (infer U)[] ? DeepReadonly<U>[] :
        T[P] extends object ? DeepReadonly<T[P]> :
        readonly T[P];
};
```

Mapped Types

```
type DeepReadonly<T> = {
    [P in keyof T]:
        T[P] extends (infer U)[] ? DeepReadonly<U>[] :
        T[P] extends object ? DeepReadonly<T[P]> :
        readonly T[P];
};
```

Mapped Types

```
type DeepReadonly<T> = {
    [P in keyof T]:
        T[P] extends (infer U)[] ? DeepReadonly<U>[] :
        T[P] extends object ? DeepReadonly<T[P]> :
        readonly T[P];
};
```

Ссылочки

- TypeScript Handbook. Advanced.

- TypeScript Deep Dive
- Андрей Старовойт — Эволюция TypeScript
- TypeScript Playground

Вопросы?

Спасибо!