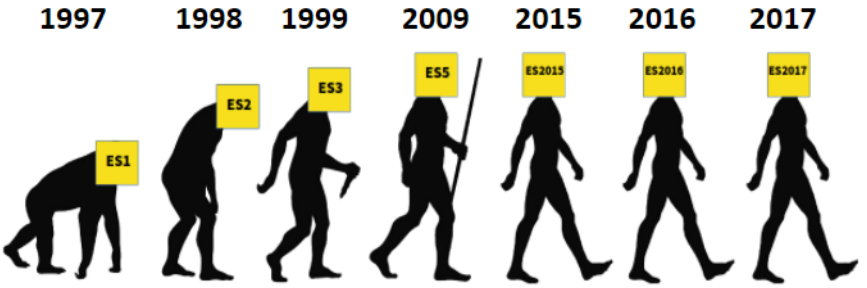
TypeScript

Старков Дима

Mocha / LiveScript / JavaScript

ECMAScript


1997
1998
1999
2009
2015
2016
2017
ES1
ES2
ES3
ES5
ES2015
ES2016
ES2017

Эволюция ECMAScript

ES1, ES2 — первые редакции

ES3 (1999) — RegExp, try/catch

ES4 — Темные времена

ES5 (2009) — строгий режим, геттеры и сеттеры, нативная поддержка JSON, новые методы Array и Object

Пример кода на ES5

```
function Student(firstName, lastName) {
    this.firstName = firstName;

    this.lastName = lastName;
}

Object.defineProperty(Student.prototype, 'name', {
    get: function() {
        return this.firstName + ' ' + this.lastName;
    }
});
```

ES6

~~ES6~~ ECMAScript 2015

Классы

let и const

Стрелочные функции

Шаблонные строки

Деструктуризация

Promise

ECMAScript 2016

Оператор ** (возведение в степень)

Array.prototype.includes

ECMAScript 2017

async / await

Object.values

Object.entries

String.prototype.pad[Start|End]

ECMAScript 2018

Rest и Spread оператор массивов

Rest и Spread оператор объектов

Promise.prototype.finally

Дальнейшее развитие ECMAScript

tc39/ecma262

Зачем всё это знать?

Статистика использования браузеров

Google Chrome	95 506 767 374	40,51 %
Яндекс.Браузер	47 186 148 692	20,02 %
Safari	25 134 247 834	10,66 %
Opera	12 329 612 387	5,23 %
Firefox	9 985 290 466	4,24 %
Android Browser	7 088 843 990	3,01 %
Samsung Internet	6 218 626 156	2,64 %
Internet Explorer	4 279 508 750	1,82 %
Edge	3 612 438 677	1,53 %

Данные из отчёта «Яндекс.Радара» на 3 декабря 2019

Таблица совместимости

https://kangax.github.io/compat-table

		Compilers/polyfills					Desktop browsers										Servers/runtimes						Mobile	
	78%	6%	56%	52%	55%	10%	0%	52%	58%	78%	78%	78%	100%	100%	83%	90%	2%	19%	77%	100%	8%	94%	83%	90%
Feature name	Current browser	Traceur	Babel 6 + core-js	Closure 2018.10	Type-Script + core-js	es7-shim	IE 11	Edge 17	Edge 18	FF 60 ESR	FF 62	FF 63	CH 69, OP 56	CH 70, OP 57	SF 11.1	SF 12	PJS	Node >=6.5 <7[2]	Node >=8.10 <9[2]	Node >=10.13 <11[2]	DUK 2.2	GraalVM 1.0[3]	iOS 11.3	iOS 12
2016 features																								
exponentiation (**) operator	3/3	2/3	3/3	3/3	2/3	0/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	0/3	0/3	3/3	3/3	2/3	3/3	3/3	3/3
Array.prototype.includes	3/3	0/3	3/3	2/3	3/3	2/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	0/3	3/3	3/3	3/3	0/3	3/3	3/3	3/3
2016 misc																								
generator functions can't be used with "new" [7]	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
generator throw() caught by inner generator [8]	Yes	No	No	Yes	Yes[9]	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
strict fn w/ non-strict non-simple params is error[10]	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
nested rest destructuring, declarations [11]	Yes	No	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
nested rest destructuring, parameters[12]	Yes	No	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Proxy, "enumerate" handler removed [13]	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Proxy internal calls, Array.prototype.includes	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
2017 features																								
Object static methods	4/4	0/4	4/4	3/4	4/4	3/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	0/4	0/4	4/4	4/4	0/4	4/4	4/4	4/4
String padding	2/2	0/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	2/2	2/2	0/2	2/2	2/2	2/2
trailing commas in function syntax	2/2	0/2	2/2	2/2	2/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	2/2	2/2	0/2	2/2	2/2	2/2
async functions	15/15	3/15	3/15	9/15	8/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	0/15	0/15	15/15	15/15	0/15	13/15	15/15	15/15
shared memory and atomics	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	17/17	17/17	0/17	0/17	0/17	0/17	17/17	17/17	0/17	17/17	0/17	0/17
2017 misc																								
Proxy "ownKeys" handler, duplicate keys for non-extensible targets (ES 2017 semantics) [22]	No	No	No	No	No	No	No	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
RegExp "u" flag, case folding	Yes	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	Yes	Yes	Yes
arguments.caller removed	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	Yes	Yes	Yes
2017 annex b																								
Object.prototype getter/setter methods	16/16	0/16	16/16	0/16	16/16	0/16	8/16	14/16	14/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	12/16	10/16	16/16	16/16	16/16	16/16	16/16	16/16
Proxy internal calls, getter/setter methods	4/4	0/4	0/4	0/4	0/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	0/4	0/4	4/4	4/4	0/4	4/4	4/4	4/4
assignments allowed in for-in head in non-strict mode	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Разный уровень поддержки языка

Транспиляция — это процесс перевода исходного кода одного языка в другой

Babel Playground

Код на ES2015

```
class Student {
    constructor(firstName, lastName) {

        this.firstName = firstName;
        this.lastName = lastName;
    }

    set skills(skills = []) {
        this._skills = skills;
    }
}
```

Код после транспиляции

```
var Student = function () {
    function Student(firstName, lastName) {
        _classCallCheck(this, Student);

        this.firstName = firstName;
        this.lastName = lastName;
    }

    _createClass(Student, [{
        key: 'skills',
        set: function() {
            var args = arguments;
            var skills = args.length > 0 && args[0] !== undefined ? args[0] : [];

            this._skills = skills;
        }
    }]);

    return Student;
}();
```

Система плагинов

Plugins

arrow-functions

classes

destructuring

duplicate-keys

for-of

С поддержкой браузерами справились

Что дальше?

Что не так?

```
function sum(a, b) {
    return a + b;

}

sum(2, 2); // 4

sum(2, '2'); // '22'

sum([], []); // ''

sum([], {}); // '[object Object]'

sum({}, []); // 0
```

Как исправить?

```
function sum(a, b) {
    if (typeof a !== 'number' || typeof b !== 'number') {

        throw new Error('a and b must be numbers');
    }

    return a + b;
}
```

Что в IDE?

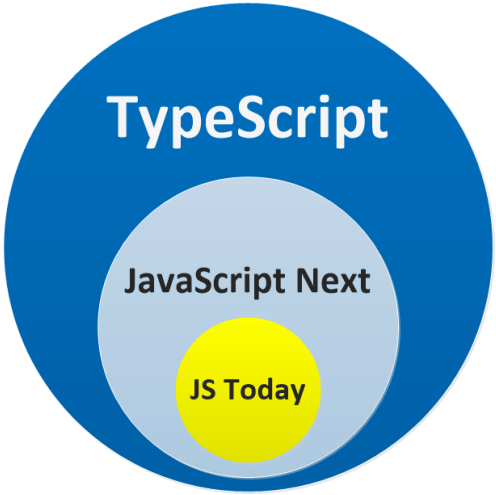
```
1  function sum(a, b) {
2      a.
3      constructor (Object, built-in)                          Function
4      hasOwnProperty(v: PropertyKey) (Object, built-i…         boolean
5      isPrototypeOf(v: Object) (Object, built-in)              boolean
6      propertyIsEnumerable(v: PropertyKey) (Object, b…         boolean
7      toLocaleString() (Object, built-in)                       string
8  }   toString() (Object, built-in)                             string
9      valueOf() (Object, built-in)                              Object
       result (-2018/…/core.js)
       $current (jquery.textillate.js)
       $texts
       _ (k, -2018/…/core.js)                                    string
       done (lib/ /head.min.js)                                  number
       Not all variants are shown, please type more letters to see the rest  Next Tip
```

Слабая динамическая типизация

Неявные преобразования

Типы становятся известны в runtime

TypeScript



TypeScript Playground

Автор

Андерс Хейлсберг

Известен также по Turbo Pascal, Delphi, C#

Возможности

ES2015-ESNext

Транспиляция в ES2015, ES5, ES3 ...

Аннотации типов и их проверка

Вывод типов

Дополнительные фичи

Примитивные типы данных

```
const isDone: boolean = false;


const decimal: number = 6 + 0xf00d + 0b1010;

const sentence: string = 'Hello, my name is Alice.';

const u: undefined = undefined;
const n: null = null;
```

Массивы (Array)

```
const list1: number[] = [1, 2, 3];


const list2: Array<number> = [1, 2, 3];
```

Кортежи (Tuple)

~~Кортежи~~ Tuple

```
// Объявление кортежа
let point: [number, number];


// Некорректная инициализация
// Type 'string' is not assignable to type 'number'.
point = [10, 'hello'];

// Инициализация
point = [20, 10]; // OK
```

Перечисления (Enum)

```
enum Color {
    Red,

    Green,
    Blue
}

const c: Color = Color.Green;
```

Перечисления (Enum)

```
enum Color {
    Red = 5,

    Green = 7,
    Blue = 9
}

const c: Color = Color.Green;
```

Перечисления (Enum)

```
enum Color {
    Red = '#f00',

    Green = '#0f0',
    Blue = '#00f'
}

const c: Color = Color.Green;
```

Объекты (Object)

```
const colors: object = {
    red: '#F00',

    green: '#0F0',
    blue: '#00F'
};
```

```
const settings: {
    color: string;

    delay: number;
    retry: boolean;
} = {
    color: '#F00',
    delay: 2000,
    retry: false
};
```

Any

```
let notSure: any = 4;
notSure = "maybe a string instead";

notSure = false;

let notSure: any = 4;
notSure.foo(); // TypeError: notSure.foo is not a function
notSure.toFixed(); // OK

let prettySure: object = new Number(4);
// Property 'toFixed' does not exist on type 'object'.
prettySure.toFixed();
```

Void и Never

```
function warnUser(): void {
    console.log("This is my warning message");

}

function error(message: string): never {
    throw new Error(message);
}

function infiniteLoop(): never {
    while (true) {}
}
```

Приведение типов

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


const strLength1: number = (<string>someValue).length;
const strLength2: number = (someValue as string).length;
```

Функции

```
function sum(x: number, y: number): number {
    return x + y;

}

const sum = (x: number, y: number): number => x + y;
```

Интерфейсы

```
function printLogin(user: { login: string }): void {
    console.log(user.login);

}

const user = {
    login: 'maxie'
};

printLogin(user);
```

Интерфейсы

```
interface IUser {
    login: string;

}

function printLogin(user: IUser): void {
    console.log(user.login);
}

const user = {
    username: 'maxie'
};

printLogin(user);
```

Опциональные свойства

```
interface IOptions {
    color?: string;
}


interface ISquare {
    color: string;
}

function create(options: IOptions): ISquare {
  const square: ISquare = { color: 'white' };

  // Type 'undefined' is not assignable to type 'string'.
  square.color = options.color;


  return square;
}
```

Опциональные свойства

```
interface IOptions {
    color?: string;
}


interface ISquare {
    color: string;
}

function create(options: IOptions): ISquare {
  const square: ISquare = { color: 'white' };

  if (options.color) {
      square.color = options.color;
  }

  return square;
}
```

Классы

```
interface IMakesSound {
    makeSound(): void;
}

class Python implements IMakesSound {
    private readonly _length: number;

    constructor(length: number) {
        this._length = length;
    }

    public get length(): number {
        return this._length / 100;
    }

    protected makeSound() {
        console.log('Ssssss!');
    }
}
```

Классы

```
abstract class Snake {
    private readonly _length: number;

    public get length(): number {
        return this._length / 100;
    }

    constructor(length: number) {
        this._length = length;
    }

    protected abstract makeSound(): void;
}
```

Классы

```
class Python extends Snake {
    private static population = 10000;

    public static incrementPopulation(): void {
        Python.population++;
    }

    constructor(length: number) {
        super(length);
        Python.incrementPopulation();
    }

    protected makeSound(): void {
        console.log('Ssssss!');
    }
}
```

Вывод типов

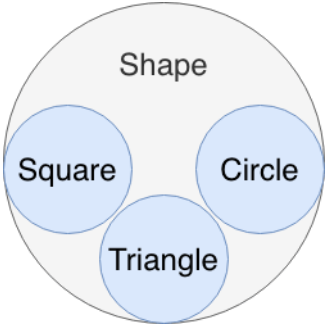
```
let n: number = 42

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

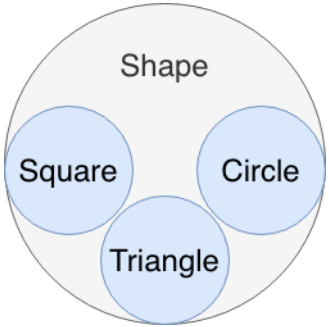
```
let shapes = [new Circle(), new Square()]
```

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



Наиболее общий тип

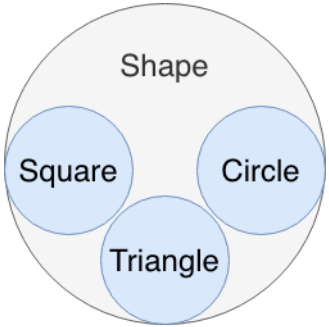
```
let shapes = [new Circle(), new Square()]


// 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()]
```

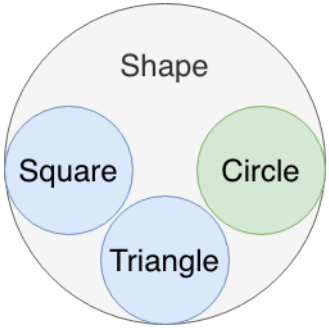
```
shapes.push(new Triangle())
```



Наиболее общий тип

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

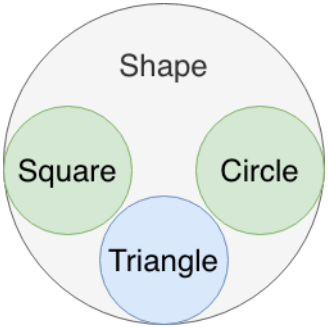
```
shapes.push(new Triangle())
```



Наиболее общий тип

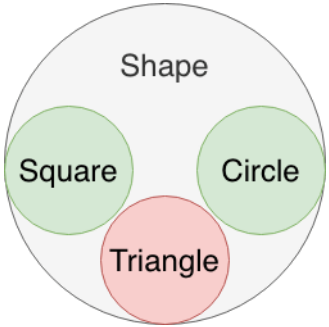
```
let shapes = [new Circle(), new Square()]
```

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



Наиболее общий тип

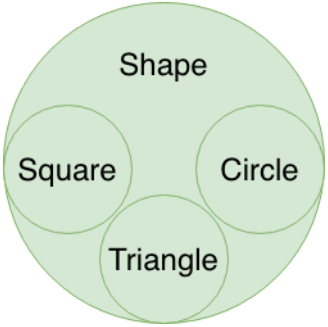
```
let shapes = [new Circle(), new Square()]


// 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')
}
```

Нужно уметь обрабатывать значения разных типов идентичным образом

Кажется нам нужен...


any
Это полиморфизм?
百科事典

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'
```

Перерыв?

TypeScript?

Спасет от выстрелов себе в ногу

ESNext прямо сейчас

Средство против TypeError

Пишет код за вас

Документация к коду

Но...

Много дополнительного кода?

Нас спасет вывод типов!

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[]
```

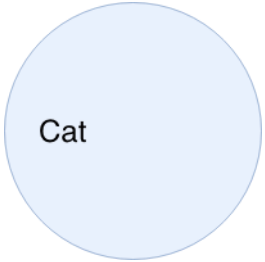
```
// String.split
split(separator: string | RegExp, limit: number): string[]
```

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

Решение: Union Types

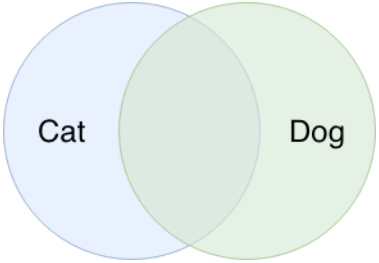
Intersection Types

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



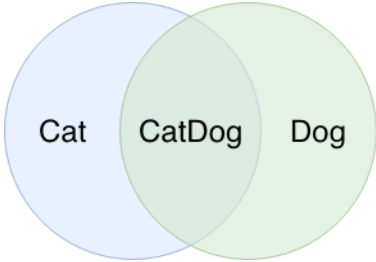
Intersection Types

```
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 – не интерфейс!

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]
}
```

А вот и нет!

```
function get(obj: any, keyName: string): any {

    return obj[keyName]
}

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

Что делать?

А вот и нет!

```
function get(obj: any, keyName: string): any {

    return obj[keyName]
}

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

Может быть любым – Generics?

А вот и нет!

```
function get(obj: any, keyName: string): any {

    return obj[keyName]
}

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

`keyName ∈ Object.keys(obj) – ?`

А вот и нет!

```
function get(obj: any, keyName: string): any {

    return obj[keyName]
}

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

`obj[keyName]` – ?

Хотим знать список полей объекта
и типы значений 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')
```

А что там в es5?

```
interface IUser {

    login: string
    age: number
    gender: 'male' | 'female'
}

const user = { login: 'dimastark', age: 22, 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: 22, 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: 22, 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: 22, 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
```

infer

```
type ReturnType<T> = T extends (

    (...args: any) => infer R
) ? R : never;

ReturnType<(a: number) => string>  // string
ReturnType<(s: string) => number>  // number
ReturnType<{ a: number }>          // never
```

Mapped Types

```
interface IUser {
    login: string

    birthDate: {
        year: number
        month: number
        day: number
    }
    gender: 'male' | 'female'
}
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    array: { s: string }[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    array: { s: string }[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: { s: string }[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: { s: string }[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: { s: string }[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: { s: string }[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: DeepReadonly<{ s: string }>[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: DeepReadonly<{ s: string }>[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: DeepReadonly<{ readonly s: string }>[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: DeepReadonly<{ readonly s: string }>[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: DeepReadonly<{ readonly s: string }>[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: DeepReadonly<{ readonly s: string }>[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: DeepReadonly<{ readonly s: string }>[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: { readonly s: string }[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

DeepReadonly<{
    readonly array: { readonly s: string }[]
}>
```

Mapped Types

```
type DeepReadonly<T> = {

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

{
    readonly array: { readonly s: string }[]
}
```

Utility Types

Pick – выбираем поля объекта

```
Pick<{
    a: string;
    b: number;
    c: boolean;
}, 'a' | 'b' | 'd'>
```

```
{
    a: string;

    c: boolean;
}
```

Utility Types

Omit – исключаем поля объекта

```
Omit<{
    a: string;
    b: number;
    c: boolean;
}, 'a' | 'b' | 'd'>
```

```
{

    c: boolean;
}
```

Utility Types

Partial – делаем все поля опциональными

```
Partial<{
    a: string | undefined;
    b: number;
    c?: boolean;
}>
```

```
{
    a?: string | undefined;
    b?: number;
    c?: boolean;
}
```

Utility Types

Required – делаем все поля обязательными

```
Required<{
    a: string;
    b?: number | undefined;
    c?: boolean;
}>
```

```
{
    a: string;
    b: number | undefined;
    c: boolean;
}
```

Utility Types

Readonly – делаем все поля readonly

```
Readonly<{
    a: string;
    b: number;
    c?: boolean;
}>
```

```
{
    readonly a: string;
    readonly b: number;
    readonly c?: boolean;
}
```

Utility Types

ReturnType – тип возвращаемого значения

```
ReturnType<() => number>        number
ReturnType<() => string>        string
ReturnType<() => boolean>       boolean
```

Utility Types

Parameters – тип аргументов функции

```
Parameters<(s: string | RegExp) => void>      [string | RegExp]
Parameters<(a: number, b: number) => void>    [number, number]
Parameters<(...nums: number[]) => void>       number[]
Parameters<(...args: any[]) => void>          any[]
```

Type Guards

Условие, дающее гарантию о безопасном сужении типа

Union Type Guard

```
function negate(n: string | number | boolean) {

    if (typeof n === 'string') {
        return '-'.concat(n);
    } else if (typeof n === 'number') {
        return -n;
    } else {
        return !n;
    }
}
```

Union Type Guard

```
function negate(n: string | number | boolean) {

    if (typeof n === 'string') {
        return '-'.concat(n);
    } else if (typeof n === 'number') {
        return -n;
    } else {
        return !n;
    }
}
```

Union Type Guard

```
function negate(n: string | number | boolean) {

    if (typeof n === 'string') {
        return '-'.concat(n);  // n is string
    } else if (typeof n === 'number') {
        return -n;
    } else {
        return !n;
    }
}
```

Union Type Guard

```
function negate(n: string | number | boolean) {

    if (typeof n === 'string') {
        return '-'.concat(n);
    } else if (typeof n === 'number') {
        return -n;  // n is number
    } else {
        return !n;
    }
}
```

Union Type Guard

```
function negate(n: string | number | boolean) {

    if (typeof n === 'string') {
        return '-'.concat(n);
    } else if (typeof n === 'number') {
        return -n;
    } else {
        return !n;  // n is boolean
    }
}
```

Union Type Guard

```
function negate(n: string | number | boolean) {

    if (typeof n === 'string') {
        return '-'.concat(n);
    } else if (typeof n === 'number') {
        return -n;
    }

    return !n;  // n is boolean
}
```

instanceof Type Guard

```
function addShape(shapes: Shape[], obj: object) {

    if (obj instanceof Shape) {
        shapes.push(obj)
    }

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

instanceof Type Guard

```
function addShape(shapes: Shape[], obj: object) {

    if (obj instanceof Shape) {
        shapes.push(obj)
    }

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

instanceof Type Guard

```
function addShape(shapes: Shape[], obj: object) {

    if (obj instanceof Shape) {
        shapes.push(obj)
    }

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

in Type Guard

```
function checkProp(obj: object, name: string): boolean {

    if (name in obj) {

        return true;
    }

    throw new TypeError(`"${name}" property is missing!`)
}
```

in Type Guard

```
function checkProp(obj: object, name: string): boolean {

    if (name in obj) {

        return true;
    }

    throw new TypeError(`"${name}" property is missing!`)
}
```

in Type Guard

```
function checkProp(obj: object, name: string): boolean {

    if (name in obj) {
        console.log(obj[name]);
        return true;
    }

    throw new TypeError(`"${name}" property is missing!`)
}
```

User Defined Type Guard

User Defined Type Guard

```
type Circle = { r: number };
type Square = { a: number };


function area(o: Circle | Square): number {
    return 'r' in o
        ? Math.PI * o.r * o.r
        : o.a * o.a;
}
```

User Defined Type Guard

```
type Circle = { r: number };
type Square = { a: number };


function area(o: Circle | Square): number {
    return 'r' in o
        ? Math.PI * o.r * o.r
        : o.a * o.a;
}
```

User Defined Type Guard

```
type Circle = { r: number };
type Square = { a: number };


function area(o: Circle | Square): number {
    return 'a' in o
        ? o.a * o.a
        : Math.PI * o.r * o.r;
}
```

User Defined Type Guard

```
type Circle = { r: number };
type Square = { a: number };


function area(o: Circle | Square): number {
    return 'a' in o
        ? o.a * o.a
        : Math.PI * o.r * o.r;
}

function isCircle(o: any): o is Circle {
    return 'r' in o && typeof o.r === 'number';
}

function isSquare(o: any): o is Square {
    return 'a' in o && typeof o.a === 'number';
}
```

User Defined Type Guard

```
type Circle = { r: number };
type Square = { a: number };


function area(o: Circle | Square): number {
    return 'a' in o
        ? o.a * o.a
        : Math.PI * o.r * o.r;
}

function isCircle(o: any): o is Circle {
    return 'r' in o && typeof o.r === 'number';
}

function isSquare(o: any): o is Square {
    return 'a' in o && typeof o.a === 'number';
}
```

User Defined Type Guard

```
type Circle = { r: number };
type Square = { a: number };


function area(o: Circle | Square): number {
    return isSquare(o)
        ? o.a * o.a
        : Math.PI * o.r * o.r;
}

function isCircle(o: any): o is Circle {
    return 'r' in o && typeof o.r === 'number';
}

function isSquare(o: any): o is Square {
    return 'a' in o && typeof o.a === 'number';
}
```

User Defined Type Guard

```
type Circle = { r: number };
type Square = { a: number };


function area(o: Circle | Square): number {
    return isCircle(o)
        ? Math.PI * o.r * o.r
        : o.a * o.a;
}

function isCircle(o: any): o is Circle {
    return 'r' in o && typeof o.r === 'number';
}

function isSquare(o: any): o is Square {
    return 'a' in o && typeof o.a === 'number';
}
```

Optional Chaining

```
let s;


s = (  // 👎
    object
    && object.nested
    && object.nested.array
    && object.nested.array[0]
    && object.nested.array[0].toString
    && object.nested.array[0].toString()
);

s = object?.nested?.array?.[0]?.toString?.();  // 👍
```

Nullish Coalescing

```
let value = options.value || 'default';


// 👎
value = 0 || 'default';
value = '' || 'default';
value = false || 'default';

// 👍
value = 0 ?? 'default';
value = '' ?? 'default';
value = false ?? 'default';
```

Ссылки

TypeScript Playground

TypeScript Handbook

TypeScript Deep Dive

Андрей Морозов (Яндекс) — Типизация

TypeScript – Тьюринг полная система типов

Андрей Старовойт (JetBrains) — Эволюция TypeScript

Спасибо!

Вопросы?