

# TypeScript

И с чем его едят

Tinkoff Fintech School 2022

# План

- Что такое TypeScript и зачем он нужен?
- Встроенные типы
- Type assertions
- Как писать свои типы
- Не забываем про duck-typing
- Generic Types
- Union типы и Type Guards
- Utility Types

# План

- Что такое TypeScript и зачем он нужен?
- Встроенные типы
- Type assertions
- Как писать свои типы
- Не забываем про duck-typing
- Generic Types
- Union типы и Type Guards
- Utility Types

О, сэр, вы из  
Англии?



# JavaScript — тиปизированный язык но не в обычном понимании

# Зачем нам TypeScript?

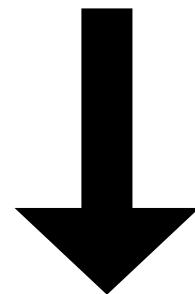
# Пример

```
function createUser(name) {  
    const user = { name };  
  
    return user;  
}
```

```
function main() {
    let user = createUser( name: 'Oleg');
    // Что есть у user? Что возвращает createUser?
    console.log(user.surname);
}
```

```
function createUser(name) {
    return name;
}

function main() {
    let user = createUser( name: 'Oleg');
    console.log(user.name.toString()); // user.name === undefined
}
```



В результате получаем:

---

```
console.log(user.name.toString()); // user.name === undefined
```

^

```
TypeError: Cannot read property 'toString' of undefined
```

# В чём проблема?

Мы не знаем, значение какого типа  
может содержать переменная

# А TypeScript что?

```
function createUser(name) {
    const user = { name };

    return name;
}

function main() {
    let user = createUser('Oleg');
    console.log(user.name);
}
```



```
interface User {
    name: string;
}

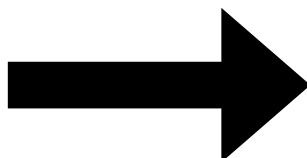
function createUser(name: string): User {
    const user: User = { name };

    return user;
}

function main() {
    let user = createUser({ name: 'Oleg' });
    console.log(user.name);
}
```

# А TypeScript что?

```
function createUser(name) {  
    const user = { name };  
  
    return name;  
}  
  
function main() {  
    let user = createUser('Oleg');  
    console.log(user.name);  
}
```

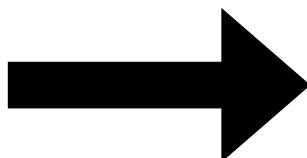


```
interface User {  
    name: string;  
}
```

```
function createUser(name: string): User {  
    const user: User = { name };  
  
    return user;  
}  
  
function main() {  
    let user = createUser({ name: 'Oleg' });  
    console.log(user.name);  
}
```

# А TypeScript что?

```
function createUser(name) {  
    const user = { name };  
  
    return name;  
}  
  
function main() {  
    let user = createUser('Oleg');  
    console.log(user.name);  
}
```

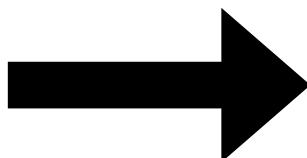


```
interface User {  
    name: string;  
}
```

```
function createUser(name: string): User {  
    const user: User = { name };  
  
    return user;  
}  
  
function main() {  
    let user = createUser({ name: 'Oleg' });  
    console.log(user.name);  
}
```

# А TypeScript что?

```
function createUser(name) {  
    const user = { name };  
  
    return name;  
}  
  
function main() {  
    let user = createUser('Oleg');  
    console.log(user.name);  
}
```

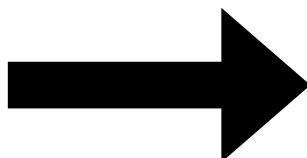


```
interface User {  
    name: string;  
}
```

```
function createUser(name: string): User {  
    const user: User = { name };  
  
    return user;  
}  
  
function main() {  
    let user = createUser({ name: 'Oleg' });  
    console.log(user.name);  
}
```

# А TypeScript что?

```
function createUser(name) {  
    const user = { name };  
  
    return name;  
}  
  
function main() {  
    let user = createUser('Oleg');  
    console.log(user.name);  
}
```



```
interface User {  
    name: string;  
}
```

```
function createUser(name: string): User {  
    const user: User = { name };  
  
    return user;  
}  
  
function main() {  
    let user = createUser({ name: 'Oleg' });  
    console.log(user.name);  
}
```

# А TypeScript что?

```
function createUser(name) {  
    const user = { name };  
  
    return name;  
}  
  
function main() {  
    let user = createUser('Oleg');  
    console.log(user.name);  
}
```



```
interface User {  
    name: string;  
}
```

```
function createUser(name: string): User {  
    const user: User = { name };  
  
    return user;  
}  
  
function main() {  
    let user = createUser({ name: 'Oleg' });  
    console.log(user.name);  
}
```

Нужен ли тип для user?

# TypeScript сам понимает, какого типа user

```
let user: User
function createUser(name: string): User {
    let user = createUser( name: 'Oleg');
    console.log(user.name);
```

Так как createUser возвращает User, мы точно знаем, что у переменной user тип User.  
Это называют type inference

# Встроенные типы

# Boolean

```
let isCool: boolean = true;  
let isBad = false;
```

# Number

```
let decimal: number = 6;  
let hex: number = 0xf00d;  
let binary: number = 0b1010;  
let octal: number = 0o744;  
let big: bigint = 100n;
```

# String

```
let color: string = "blue";
color = 'red';
```

# Array

```
let list: number[] = [1, 2, 3];
let secondList: Array<number> = [1, 2, 3];
```

# Array

```
let list: number[] = [1, 2, 3];
let secondList: Array<number> = [1, 2, 3];
```



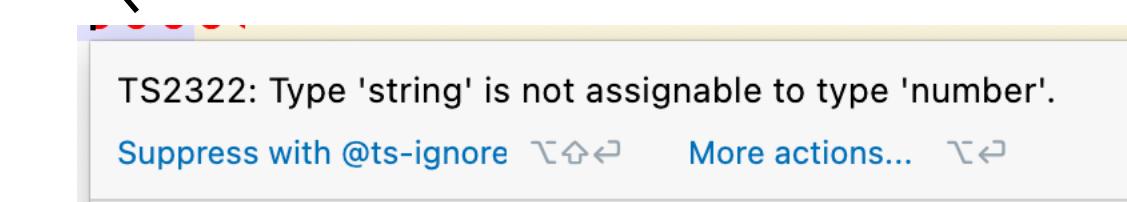
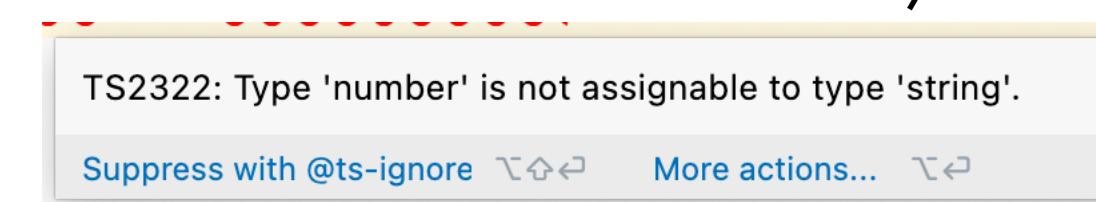
Это называется generic type. Разберем чуть дальше.

# Tuple

```
// Создем tuple
let x: [string, number];
// Инициализация
x = ["hello", 10]; // OK
// Проверка типа
x = [10, "hello"]; // Error
```

# Tuple

```
// Создаем tuple
let x: [string, number];
// Инициализация
x = ["hello", 10]; // OK
// Проверка типа
x = [10, "hello"]; // Error
```



# Enum

```
enum Color {  
    Red = 0,  
    Green = 1,  
    Blue = 2,  
}  
let c: Color = Color.Green;
```

# Enum

```
enum Color {  
    Red = 0,  
    Green = 1,  
    Blue = 2,  
}  
let c: Color = Color.Green;
```

По дефолту задаются численные значения от 0

# Enum

```
enum Color {  
    Red = 1,  
    Green = 2,  
    Blue = 'Blue',  
}  
let c: Color = Color.Green;
```

Могут быть и строчные

# Enum

```
enum Color {  
    Red = 1,  
    Green = 2,  
    Blue = 3,  
}  
let colorName: string = Color[2];  
  
// В консоли будет 'Green'  
console.log(colorName);
```

# Enum

```
enum Color {  
    Red = 1,  
    Green = 2,  
    Blue = 3,  
}  
let colorName: string = Color[2];  
  
// В консоли будет 'Green'  
console.log(colorName);
```

# Enum

```
enum Color {  
    Red = 1,  
    Green = 2,  
    Blue = 3,  
}  
let colorName: string = Color[2];  
  
// В консоли будет 'Green'  
console.log(colorName);
```

Это работает, потому что:

```
Color[2] === 'Green'  
Color['Green'] === 2;
```

# Enum

```
enum Color {  
    Red = 1,  
    Green = 2,  
    Blue = 3,  
}  
let colorName: string = Color[2];  
  
// В консоли будет 'Green'  
console.log(colorName);
```

Это работает, потому что:

IDE нас остерегает, что так  
лучше не делать

Color[2] === 'Green'  
Color['Green'] === 2;

# Enum: осторожно!

```
enum Color {  
    Red = 1,  
    Green = 2,  
    Blue = 2,  
    Yellow = 3  
}
```

```
Color[2] === 'Blue';  
Color['Blue'] === 2;  
Color['Green'] === 2;
```

# Any

```
declare function getValue(key: string): any;  
// Результат getValue может быть чем угодно  
const str: string = getValue(key: "myString");
```

Any нужно использовать только в безвыходных ситуациях

# Unknown

```
let notSure: unknown = 4;  
notSure = "maybe a string instead";  
  
// Ну а теперь точно boolean  
notSure = false;
```

Как работать с такими «непонятными» типами разберем дальше

# Any vs Unknown

```
const unknownValue: unknown = 'this is actually a string';
const something: string = unknownValue;
```

```
const anotherValue: any = 'this is actually a string';
const anotherSomething: string = anotherValue;
```

# Any vs Unknown

```
const unknownValue: unknown = 'this is actually a string';
const something: string = unknownValue;
```

Это не сработает (ну и отлично)

```
const anotherValue: any = 'this is actually a string';
const anotherSomething: string = anotherValue;
```

А тут TypeScript не пожалуется

# Void

```
function myFunc(): void {  
    console.log("Я такая хорошая и ничего не возвращаю");  
}
```

Void существует, чтобы показать, что функция ничего не возвращает, и что не надо её результат ни к чему приравнивать

# Void

Бесполезен для переменных, так как можно присвоить только null или undefined  
(если не включен strictNullCheck)

```
let unusable: void = undefined;  
// Если нет strictNullCheck  
unusable = null;
```

# Null и Undefined

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

Но если не включен strictNullCheck:

```
let n: number = null;  
let c: object = null;  
let p: string = undefined;
```

В других языках это называется «выключить null-safety»

# Never

```
// Функция никогда не завершится нормально, без ошибки
function error(message: string): never {
    throw new Error(message);
}

// Type inference определит тип как never
function fail() {
    return error("Something failed");
}

// Функция никогда не завершится, значит и не сможет ничего вернуть
function infiniteLoop(): never {
    while (true) {}
}
```

Важно заметить, что даже any нельзя приравнять к never

# Object

```
declare function create(o: object): void;

// OK
create(o: { prop: 0 });

// OK если отключен strictNullCheck
create(o: null);
create(o: undefined); // with `--strictNullChecks` flag enabled, undefined is not a subtype of null

// Не сработает
create(o: 42);
create(o: "string");
create(o: false);
```

Все «непримитивы» приравниваются к object

Не используйте Object без необходимости  
Это еще один Anу «на минималках»

# Type Assertion

```
// Вариант 1
let someValue: unknown = "this is a string";

let strLength: number = (someValue as string).length;

// Вариант 2
let someOtherValue: unknown = "this is a string";

let someOtherLength: number = (<string>someValue).length;
```

Так делать тоже лучше не стоит, если в этом нет необходимости

Как более элегантно определить тип в someValue рассмотрим в главе «Type Guards»

# Как написать свой тип?

# С помощью type

```
|type Point = {  
    x: number;  
    y: number;  
}  
  
const point: Point = { x: 1, y: 2};  
const notAPoint: Point = {x: 3};
```

# С помощью type

```
]type Point = {  
    x: number;  
    y?: number;  
}  
  
const point: Point = { x: 1, y: 2};  
const stillAPoint: Point = {x: 3};  
const notAPoint: Point = {y: 4};
```

# С помощью type

```
]type Point = {  
    x: number;  
    Optional, необязательное поле y?: number;  
}  
  
const point: Point = { x: 1, y: 2};  
const stillAPoint: Point = {x: 3};  
const notAPoint: Point = {y: 4};
```

# С помощью type

```
type Point = {  
    x: number;  
    y?: number;  
    rotate(): void;  
    anotherRotate: () => void;  
    rotateWithParams(angle: number): void;  
    anotherRotateWithParams: (angle: number) => void;  
}
```

# С помощью Interface

```
interface Point {  
    x: number;  
    y?: number;  
    rotate(): void;  
    anotherRotate: () => void;  
    rotateWithParams(angle: number): void;  
    anotherRotateWithParams: (angle: number) => void;  
}
```

Стоп, а в чем разница?

Ответ тут

# Разница 1: extends

```
interface Person {  
    name: string;  
}  
  
interface Employee extends Person {  
    job: string;  
}  
  
function acceptPersons(person: Person): void {  
    console.log(`I am a person and my name is ${person.name}`);  
}  
  
const worker: Employee = { name: 'Poor Guy', job: 'CTO' };  
acceptPersons(worker);
```

Типы тоже умеют, но немного иначе

# Разница 1: intersection types

```
type Person = {
    name: string
}

type Employee = Person & {
    job: string;
}

function acceptPersons(person: Person): void {
    console.log(`Still a person with a name ${person.name}`);
}

const person: Employee = {name: 'John', job: 'CEO'};
acceptPersons(person);
```

Intersection объединяет несколько  
типов в один

# Разница 2: повторное определение

```
interface Window {  
    title: string;  
}
```

```
interface Window {  
    custom: string;  
}
```

```
console.log(window.title);  
console.log(window.custom);
```

```
type Window = {  
    title: string;  
}
```

```
type Window = {  
    custom: string;  
}
```

```
console.log(window.title);  
console.log(window.custom);
```

Интерфейсы объединяются при повторном определении, типы — нет, они кидают ошибку.

# А что использовать?

# Duck-typing

# Duck-typing

```
type User = {  
    name: string;  
}  
  
function onlyAcceptUsers(user: User): void {  
    console.log(`Hi, ${user.name}`);  
}  
  
const notAUser = { name: 'Gosha', age: 93 };  
  
onlyAcceptUsers(notAUser);
```

# Duck-typing

```
type User = {  
    name: string;  
}  
  
function onlyAcceptUsers(user: User): void {  
    console.log(`Hi, ${user.name}`);  
}  
  
const notAUser = { name: 'Gosha', age: 93 };  
  
onlyAcceptUsers(notAUser);
```

Ошибки не будет

**‘If it walks like a duck and it quacks like a duck,  
then it must be a duck’**

**Тип переменной != тип значения**

# Duck-typing

```
type User = {
    name: string;
}

type AlmostUser = {
    name?: string;
    age: number;
}

function onlyAcceptUsers(user: User): void {
    console.log(`Hi, ${user.name}`);
}

const notAUser: AlmostUser = { name: 'Gosha', age: 93 };

onlyAcceptUsers(notAUser);
```

Это правильная логика, здесь ошибки TypeScript нет

# Полезные способы комбинировать типы

# Generic Types

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

# Generic Types

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

Очень плохо. Не знаем, что функция принимает, а что возвращает

Нет связи между типом аргумента и возвращаемым типом

# Generic Types

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

function onlyAcceptStrings(value: string): void {
    console.log(value);
}

const firstValue = identity<string>( arg: 'some string');
const secondValue = identity<number>( arg: 1);

onlyAcceptStrings(firstValue);
onlyAcceptStrings(secondValue);
```

# Generic Types

```
const firstValue = identity(arg: 'some string');
const secondValue = identity(arg: 1);

onlyAcceptStrings(firstValue);
onlyAcceptStrings(secondValue);
```

# Generic Types

```
const firstValue = identity(arg: 'some string');
const secondValue = identity<string>(arg: 1);

onlyAcceptStrings(firstValue);
onlyAcceptStrings(secondValue);
```

# Generic Types

```
function identity<Type>(arg: Type): Type {  
    console.log(arg.length);  
  
    return arg;  
}  
  
const firstValue = identity(arg: 'some string');
```

# Generic можно ограничить

```
interface Person {  
    name: string;  
}  
  
function identity<Type extends Person>(arg: Type): Type {  
    return arg;  
}  
  
const firstValue = identity(arg: 'some string');  
const secondValue = identity(arg: { name: 'Misha' });
```

# А зачем?

```
|type Person = {  
    name: string;  
}  
  
|function identity1<T extends Person>(arg: T): T {  
    return arg;  
}  
  
|function identity2(arg: Person): Person {  
    return arg;  
}  
  
const someone = { name: 'Misha', age: 90 };  
  
const s1 = identity1(someone);  
const s2 = identity2(someone);  
  
console.log(s1.age);  
console.log(s2.age);
```

# Их может быть несколько

```
interface Person {
    name: string;
}

interface Job {
    title: string;
}

function identity<T extends Person, J extends Job>(arg: T, job: J): T {
    console.log(`Person ${arg.name} has a job of ${job.title}`);
    return arg;
}
```

# У классов все работает точно также

```
class Person<J> {  
    private readonly _job: J;  
  
    constructor(job: J) {  
        this._job = job;  
    }  
  
    get job(): J {  
        return this._job;  
    }  
}
```

# Как еще можно создавать свои из существующих?

Ответ на этот вопрос лежит тут

# Type Guards & Union Types

# Type Guards

```
/**  
 * Берем строку и добавляем паддинг в начале  
 * Если 'padding' string, тогда добавляем значение в начало  
 * Если 'padding' число, то добавляем такое кол-во пробелов в начало  
 */  
  
function padLeft(value: string, padding: any) {  
    if (typeof padding === "number") {  
        return Array(arrayLength: padding + 1).join(" ") + value;  
    }  
    if (typeof padding === "string") {  
        return padding + value;  
    }  
    throw new Error(`Ожидали строку или число, а получили '${typeof padding}'.`);  
}  
  
padLeft( value: "Hello world", padding: 4); // возвращает "      Hello world"
```

# Type Guards

```
/**  
 * Берем строку и добавляем паддинг в начале  
 * Если 'padding' string, тогда добавляем значение в начало  
 * Если 'padding' число, то добавляем такое кол-во пробелов в начало  
 */  
function padLeft(value: string, padding: any) {  
    if (typeof padding === "number") { ←  
        return Array(arrayLength: padding + 1).join(" ") + value;  
    }  
    if (typeof padding === "string") {  
        return padding + value;  
    }  
    throw new Error(`Ожидали строку или число, а получили '${typeof padding}'.`);  
}  
  
padLeft( value: "Hello world", padding: 4); // возвращает "      Hello world"
```

Это один из видов type guard-ов

# Type Guards

```
if (typeof padding === "number") {
    padding.
        return mtoFixed(fractionDigits?: number) string
    }
}
if (typeof padding === "string") {
    return mtoLocaleString(locales?: string | string[], opt... string
}
m valueOf() number
p constructor (Object, typescript, es5) Function
m toPrecision(precision?: number) string
m toExponential(fractionDigits?: number) string
m toString(radix?: number) string
d deepFreeze<T>(instance: T) (Object, aml/src/.../deep-fre...
m hasOwnProperty(v: PropertyKey) (Object, typescript, ... boolean
m isPrototypeOf(v: Object) (Object, typescript, ... boolean
m propertyIsEnumerable(v: PropertyKey) (Object, ... boolean
T as FunctionCall<any>
Press ^ to choose the selected (or first) suggestion and insert a dot afterwards Next Tip :
```

Оператор `instanceOf` и `in` тоже могут служить как guard-ы для типов

# Какая проблема?

```
padLeft( value: 'Hello' , padding: true);
```



Как ограничить принимаемые типы для padding?

# Union Type

```
function padLeft(value: string, padding: string | number) {  
  if (typeof padding === "number") {  
    return Array(arrayLength: padding + 1).join(" ") + value;  
  }  
  if (typeof padding === "string") {  
    return padding + value;  
  }  
}
```

Guard-ы позволяют в том числе работать с Union-типами, чтобы понять, значение какого конкретно типа сейчас находится в переменной

# Возможные значения

```
type myType = object | string;  
  
function myFunc(arg: myType): void {  
    console.log(arg);  
}  
  
myFunc( arg: { price: 5 } );  
myFunc( arg: 'other string' );  
myFunc( arg: 5 );
```

```
type myType = 5 | 'hello' | 'nice';  
  
function myFunc(arg: myType): void {  
    console.log(arg);  
}  
  
myFunc( arg: 'hello' );  
myFunc( arg: 'other string' );  
myFunc( arg: 5 );
```

А что же делать с нашими,  
невстроеными типами?

# Вариант 1: Дать TypeScript-у понять самому

```
interface ClickEvent {
    data: string;
    target: string;
}

interface ScrollEvent {
    length: number;
}

function onEvent(event: ClickEvent | ScrollEvent): void {
    if ('data' in event) {
        console.log(event.target);

        return;
    }

    console.log(event.length);
}
```

Такой вариант больше подходит, когда вы сами управляете тем, с какими данными работаете

# Вариант 1: Дать TypeScript-у понять самому

```
interface ClickEvent {
    data: string;
    target: string;
}

interface ScrollEvent {
    length: number;
}

function onEvent(event: ClickEvent | ScrollEvent): void {
    if ('data' in event) {    Только у ClickEvent есть data, значит это точно он
        console.log(event.target);

        return;
    }

    console.log(event.length);
}
```

Такой вариант больше подходит, когда вы сами управляете тем, с какими данными работаете

# Вариант 1: Дать TypeScript-у понять самому

```
interface ClickEvent {
    data: string;
    target: string;
}

interface ScrollEvent {
    length: number;
}

function onEvent(event: ClickEvent | ScrollEvent): void {
    if ('data' in event) { Толькo у ClickEvent есть data, значит это точно он
        console.log(event.target);

        return; Если ClickEvent, то функция завершается
    }

    console.log(event.length); Если не ClickEvent (так как при нем функция уже завершилась), то точно ScrollEvent
}
```

# Вариант 2: Discriminating Unions

```
interface ClickEvent {  
    data: string;  
    target: string;  
}
```

```
interface ScrollEvent {  
    data: string;  
    length: number;  
}
```

Тоже можно проверять через `in`, но это может быть сложно, если таких Event-ов у нас будет много

# Вариант 2: Discriminating Unions

```
interface ClickEvent {
    type: 'click';
    data: string;
    target: string;
}

interface ScrollEvent {
    type: 'scroll';
    data: string;
    length: number;
}

function onEvent(event: ClickEvent | ScrollEvent): void {
    if (event.type === 'click') {
        console.log(event.target);
    } else if (event.type === 'scroll') {
        console.log(event.length);
    }
}
```

Вместо типа прописали строчное значение

# Вариант 2: Discriminating Unions

```
interface ClickEvent {
    type: 'click';
    data: string;
    target: string;
}

interface ScrollEvent {
    type: 'scroll';
    data: string;
    length: number;
}

type Events = ScrollEvent | ClickEvent;

function onEvent(_event: Events): void {}
```

# Вариант 3: пишем свой type guard

```
interface ResponseA {
    data: string[];
}

interface ResponseB {
    data: string;
}

function processResponse(response: ResponseA | ResponseB): void {
    if (isResponseA(response)) {
        response.data.forEach(value => console.log(value));

        return;
    }

    console.log(response.data);
}

function isResponseA(response: ResponseA | ResponseB): response is ResponseA {
    return Array.isArray(response.data);
}
```

Иногда type inference может сам не справиться, тогда мы можем написать свой guard, чтобы ему помочь

Но это тоже лучше делать только в крайних случаях

# Utility Types

# Partial<Type>

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

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

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

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

Все поля становятся Optional

# Required<Type>

```
interface Props {  
    a?: number;  
    b?: string;  
}  
  
const obj: Props = { a: 5 };  
  
const obj2: Required<Props> = { a: 5 };
```

Наоборот, все поля становятся обязательными

# Readonly<Type>

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

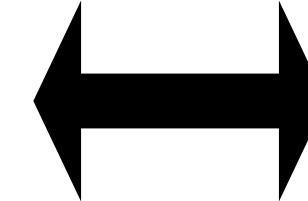
```
const todo: Readonly<Todo> = {  
    title: "Delete inactive users",  
};
```

```
todo.title = "Hello";
```

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

```
const todo: Todo = {  
    title: "Delete inactive users",  
};
```

```
todo.title = "Hello";
```



Все поля становятся readonly

# Record<Keys, Type>

```
interface CatInfo {
    age: number;
    breed: string;
}

type CatName = "miffy" | "boris" | "mordred";

const cats: Record<CatName, CatInfo> = {
    miffy: { age: 10, breed: "Persian" },
    boris: { age: 5, breed: "Maine Coon" },
    mordred: { age: 16, breed: "British Shorthair" },
};

cats.boris;
```

# Pick<Type, Keys>

```
interface Todo {  
    title: string;  
    description: string;  
    completed: boolean;  
}  
  
type TodoPreview = Pick<Todo, "title" | "completed">;  
  
const todo: TodoPreview = {  
    title: "Clean room",  
    completed: false,  
};
```

# Omit<Type, Keys>

```
interface Todo {  
    title: string;  
    description: string;  
    completed: boolean;  
    createdAt: number;  
}  
  
type TodoPreview = Omit<Todo, "description">;  
  
const todo: TodoPreview = {  
    title: "Clean room",  
    completed: false,  
    createdAt: 1615544252770,  
};
```

# Exclude<UnionType, ExcludedMembers>

```
type T0 = Exclude<"a" | "b" | "c", "a">;  
  
// type T0 = "b" | "c"  
type T1 = Exclude<"a" | "b" | "c", "a" | "b">;  
  
// type T1 = "c"  
type T2 = Exclude<string | number | (() => void), Function>;  
  
// type T2 = string | number
```

Позволяет исключать какие-то типы из type union-ов

# ReturnType<Type>

```
type T0 = ReturnType<() => string;
//type T0 = string
```

```
type T1 = ReturnType<(s: string) => void;
//type T1 = void
```

```
type T2 = ReturnType<<T>() => T>;
//type T2 = unknown
```

Позволяет исключать какие-то типы из type union-ов

И еще много всяких разных Utility  
Types описаны [здесь](#)

# Полезные ссылки

- [TypeScript Overview](#)
- [Список повседневных типов](#)
- [Union and Intersection Types](#)
- [Разница между type и interface](#)
- [Создаем типы из типов](#)
- [Utility Types](#)
- [Статья: простые TypeScript хитрости](#)

**Спасибо за внимание!**