Введение в TypeScript

Добавляем типы в JS

Повышаем надёжность кода

Кирилл Талецкий

TeachMeSkills
14 сентября 2023

Что будет

- Зачем нужна типизация
- Принцип работы TypeScript
- Базовые типы (number, string, boolean)
- Массивы, кортежи, объекты, перечисления
- Функции - введение
- Интерфейсы - свойства, расширения
- Объединения и пересечения
- Псевдонимы
- Сужение типов (Type narrowing)
  - Защитника (Type guards)
  - Различение типов (Discriminated unions)
- Перегрузка функций
- Обобщения (Generics)

Зачем нужна типизация

- Валидный код в JS

```
let value = 'some string';

// some code

value = {
  foo: 'bar'
}

// some other code

value.split(' ').join('\n');
```

Зачем нужна типизация

- Валидный код в JS

```
let value = 'some string';

// some code

value = {
  foo: 'bar'
}

// some other code

value.split(' ').join('\n');
```

```
▶ Uncaught TypeError: value.split is not a function
    at <anonymous>:11:7
```

Зачем нужна типизация

- Тот же самый код, но с проверкой типов

```
let value: string = 'some string';

// some code

value = {
  foo: 'bar'
}

// some other code

value.split(' ').join('\n');
```

Зачем нужна типизация

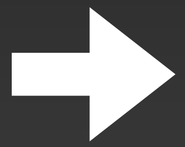
- Тот же самый код, но с проверкой типов

```
Type '{ foo: string; }' is not assignable to type 'string'. (2322)

let value: string

View Problem (⌥F8)    No quick fixes available

value = {
  foo: 'bar'
}

// some other code

value.split(' ').join('\n');
```

Пару слов о принципе работы TS

- TS - это надмножество над JS
  - Любой валидный JS код является валидным TS кодом
- TS делает статическую проверку кода
  - То есть, не запускает сам код, а использует аннотации типов для анализа
- TS компилируется в JS
  - В рантайме не будет типов, только чистый JS


```ts
let value: string = 'some string';

value.split(' ').join('\n');
```




```js
"use strict";
let value = 'some string';
value.split(' ').join('\n');
```

Базовые типы

- Чаще всего вам придётся применять:
  - number
  - string
  - boolean

```
const str: string = 'foo'
const num: number = 42
const bool: boolean = true;

const empty: null = null;
const notDefined: undefined = undefined;

const bigInt: bigint = BigInt(424242);
const sym: symbol = Symbol('bar');
```

Составные типы

Составные типы
Массивы

- Обозначение массивов строк, чисел, булевых значений:

```
let strings: string[] = ['foo', 'bar', 'baz'];
let numbers: number[] = [1, 2, 3, 4, 5];
let bools: boolean[] = [true, false, true, true];
```

- То же самое, но другое обозначение

```
let strings: Array<string> = ['foo', 'bar', 'baz'];
let numbers: Array<number> = [1, 2, 3, 4, 5];
let bools: Array<boolean> = [true, false, true, true];
```

- Обозначение вложенных массивов

```
let strings: string[][] = [
  ['foo', 'bar', '42'],
  ['bar', 'baz', 'foo'],
  ['foo', 'bar', '42']
];
```

Составные типы
Кортежи - Tuples

- Упорядоченный набор фиксированной длины
  - Задаёт порядок
  - Задаёт фиксированное количество элементов
  - Задаёт тип каждого элемента

```
let point2dWithDescription: [number, number, string] = [0, 0, 'Description: Center of coordinates']

point2dWithDescription = [0, 0];
point2dWithDescription = ['0', 0, 'Description: Center of coordinates'];
point2dWithDescription = ['Description: Center of coordinates', 0, 0];
```

  - Есть возможность указать имена каждого элемента кортежа

```
let point2dWithDescription: [x: number, y: number, description: string] = [0, 0, 'Description: Center of coordinates']
```

Объекты

- Можно строго задать названия и тип полей

```
const person: {name: string, age: number} = {
  name: 'Ivan',
  age: 42,
}
```

- Можно обозначить опциональные поля

```
const ivan: {name: string, age: number, notes?: string} = {
  name: 'Ivan',
  age: 42,
  notes: 'likes apples'
}

const rostislav: {name: string, age: number, notes?: string} = {
  name: 'Rostislav',
  age: 42,
}
```

Составные типы
Перечисления - Enums

- Набор ключ - значение
- Конечное количество

- Используется одновременно и как тип, и как объект, который будет существовать в рантами (!)
- Переменная с типом из перечисления может принимать только значения этого перечисления
- Удобно для рефакторинга большого количества кода
  - Можно поменять значения только в самом enum’е, они будут изменены во всём проекте

- Неименованные перечисления
  - Рантайм значения генерятся автоматически
  - Не рекомендуются к использованию, есть особенность компиляции в JS объект

Функции, введение

Функции
Аргументы и возвращаемые значения

- Обычная типизация аргументов и возвращаемого значения

```
function getValue(prefix: string, num: number): string {
  return `${prefix}${num}`
}
```

- Типизация анонимной функции

```
let baz: (value: string) => string

baz = (value) => {
  return value;
}
```

- Типизация произвольного количества аргументов

```
function concat(...args: string[]): string {
  return args.join('');
}
```

Функции

Бонус: хорошие практики типизации аргументов

```ts
// ❌ Не надо так

function getFullName(firstName: string, lastName?: string, middleName?: string) {
  return [firstName, lastName, middleName].filter(Boolean).join(' ');
}

// пользователь функции может легко перепутать аргументы (и обязательно это сделает)

getFullName("Kirill", "Vladimirovich", "Taletski");
```

```ts
// ✅ если аргументы однородные, лучше передавать объект, где поля явно проименованы

function getFullName({firstName, lastName, middleName}: {firstName: string, lastName?: string, middleName?: string}) {
  return [firstName, lastName, middleName].filter(Boolean).join(' ');
}

// так пользователь функции вряд ли ошибётся

getFullName({firstName: 'Kirill', lastName: 'Taletski', middleName: 'Vladimirovich'})
```

Псевдонимы типов

Псевдонимы

- “переменные” для типов
- Позволяют создавать сложные типы
- Псевдонимы типов можно комбинировать между собой

Интерфейсы

Интерфейсы

- Описывают тип обьекта
  - Задают названия ключей
  - Задают тип значений

- Все поля являются обязательными по умолчанию
  - Можно указать опциональные поля
- Тип полей можно указать другими интерфейсами
  - Тип полей можно указать через родительский интерфейс (рекурсивно)
- Поле может быть функцией
  - Типизировать можно через короткую запись или через стрелочный синтаксис

Интерфейсы
Расширение

- Интерфейсы можно расширять
  - Явно - указав один или несколько базовых интерфейсов
  - Неявно - определив несколько интерфейсов с одинаковым именем

Объединения и пересечения

Объединения
Unions

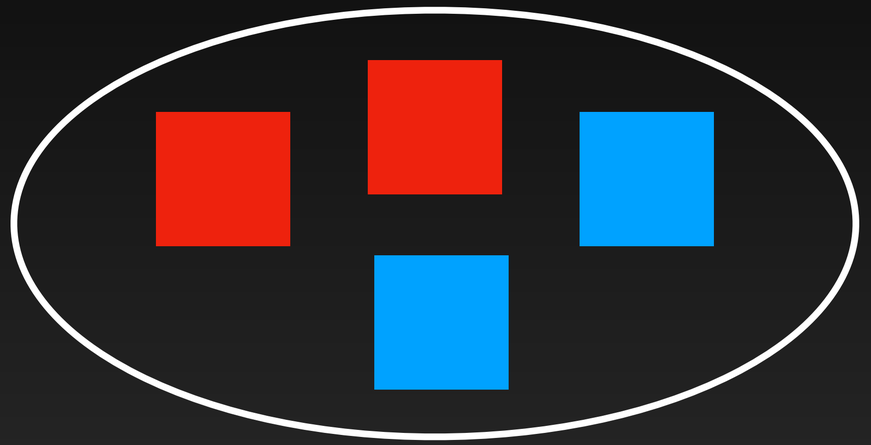
- А что делать, если мы хотим присвоить в переменную
  - Строку **ИЛИ** число?
  - Обьект с интерфейсом Person **ИЛИ** с интерфейсом Employee

- Для этого в TS есть объединения - Union Types

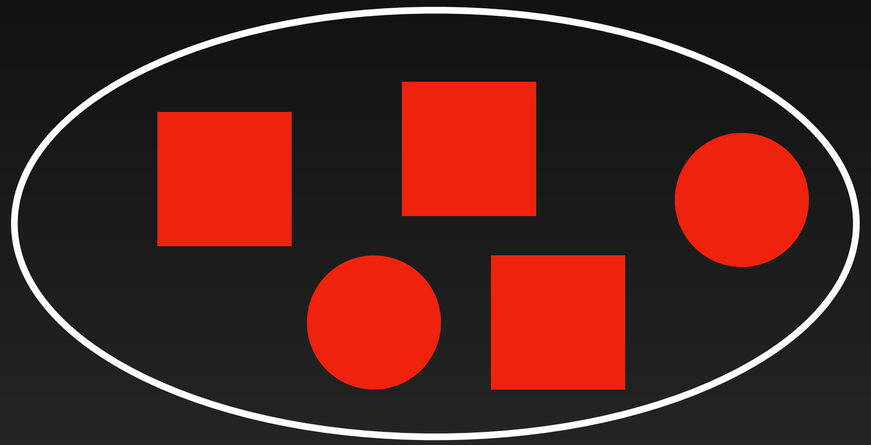
```
const price: string | number = 300;
const price2: string | number = '$300'

const someone: Person | Employee = { name: 'Ivan', age: 42 }
const stranger: Person | Employee = { name: 'Kek', age: 42, jobTitle: 'Senior HTML Coder'}
```

Объединения

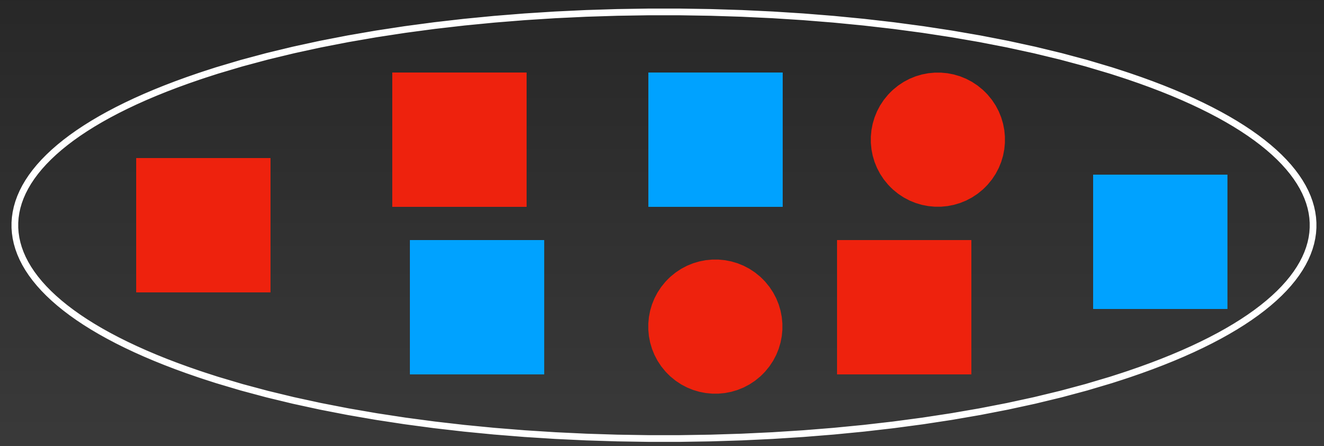
Square



Red



Square | Red

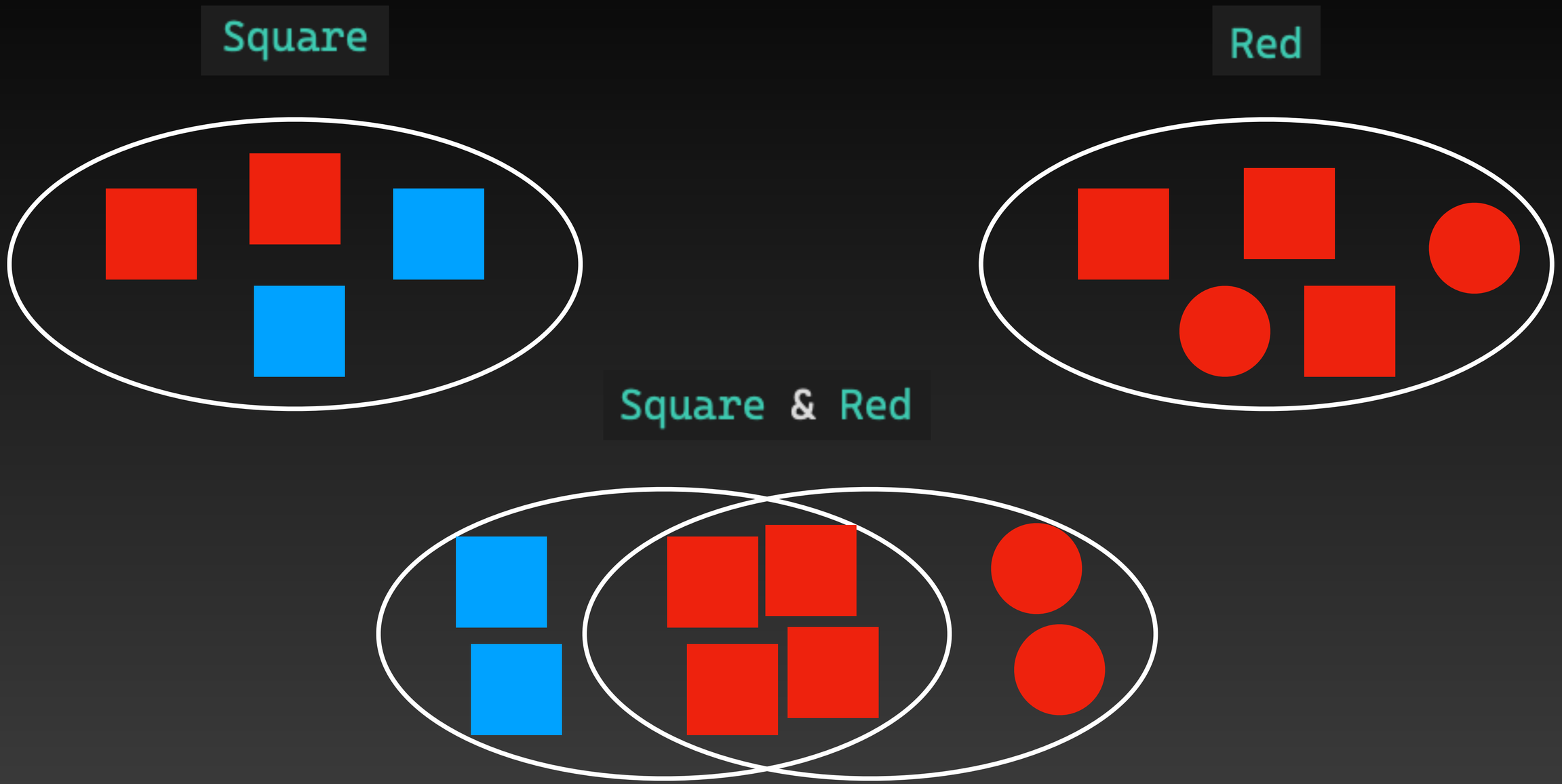


Пересечения
Intersection

- А если мы хотим сделать тип, который включает все поля нескольких типов?

```
interface Fish {
  swim(): void;
}

interface Bird {
  fly(): void;
}

const birdfish: Bird & Fish = {
  swim: () => {},
  fly: () => {}
}
```

Пересечения



Литеральные типы

Литеральные типы
Literal Types

- Позволяют создавать более специфичные ограничения типа

```
let color: 'red' = 'red';

color = 'blue';
```

- Литеральные типы очень полезны в сочетании с объединениями

```
let color: 'red' | 'blue' | 'black' | 'white'  = 'red';

color = 'blue';
```

- Можно использовать почти так же, как и перечисления, но с нюансами.

Другие простейшие типы

Другие простейшие типы
Unknown

- Позволяет обозначить “любое” возможное значение
- Представляет собой максимально широкий тип, включающий в себя все остальные
- В unknown переменную можно присвоить любой тип
- Чтобы работать с unknown переменной, её тип нужно будет сужать (об этом далее)

Другие простейшие типы

Any

- Обозначает любое возможное значение
- Полностью отключает проверку типов TS, даже для выведенных типов
- Тип any нужен для постепенной миграции JS кода на TS, и только для этого

Никогда не используйте any в продуктовом коде

- Скорее всего, для вашего случая подойдёт unknown
- Его можно использовать только в случае крайней необходимости, например:
  - Библиотека, которую вы используете, использует any... и вы не можете её пропатчить / сделать пул реквест в репозиторий

Другие простейшие типы
Never, Void

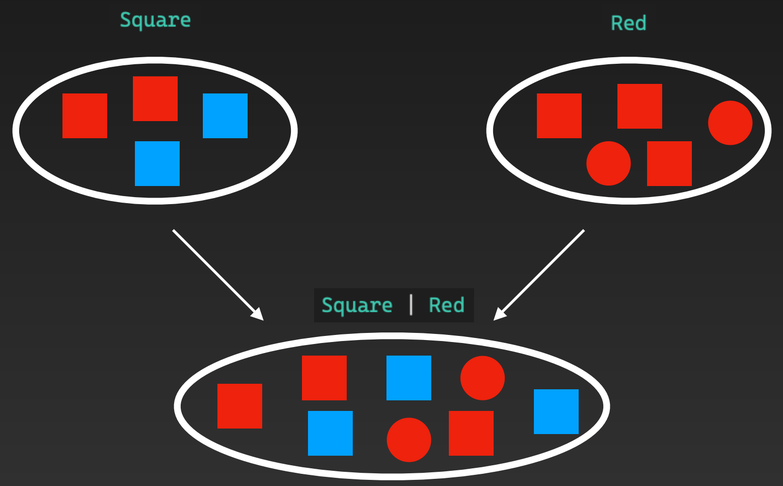
- Never чаще всего обозначает отсутствие возвращаемого значения
  - Например, если функция всегда бросает ошибку, она возвращает never
  - Пересечение двух непересекающихся типов вернёт never

- Void говорит о том, что мы неявно возвращаем undefined
  - Исторически используется для обозначения return type функций, у которых нет return (неявно возвращают undefined)

Сужение типов

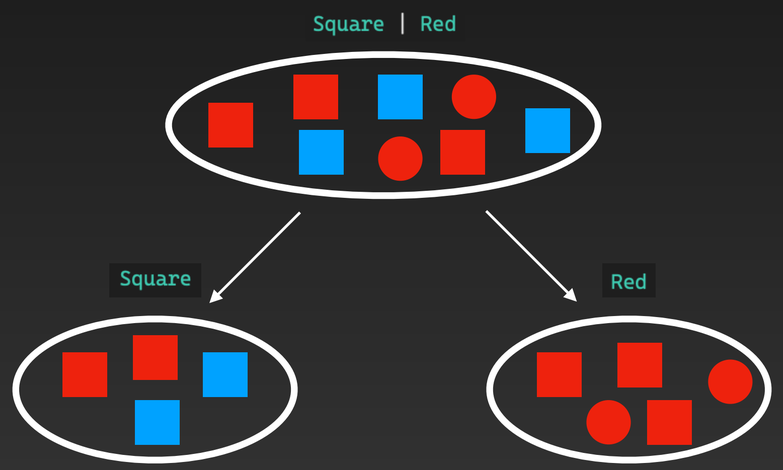
Сужение типов

- Мы с вами умеем создавать “широкие” типы, используя Union и Intersection



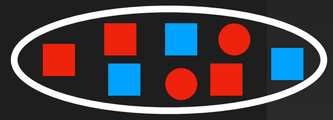
Сужение типов

- А что делать, если мы хотим сделать обратное — выделить узкий подтип из более широкого типа?



Discriminated Unions
Различение

```ts
interface Cat {
  type: 'cat'; // дискриминатор
  meow: () => void;
}

interface Dog {
  type: 'dog'; // дискриминатор
  bark: () => {}
}

function checkVoice(catOrDog: Cat | Dog): void {
  switch (catOrDog.type) {
    case 'cat':
      catOrDog.meow();
      break;
    case 'dog':
      catOrDog.bark();
      break;
  }
}
```



Type Guard

```ts
function isDog(value: unknown): value is Dog {
  return typeof value === 'object'
    && value !== null
    && 'type' in value
    && value.type === 'dog';
}

function barkIfDog(object: unknown) {
  if (isDog(object)) object.bark();
}
```

Перегрузка функций

Перегрузка функции

- Перегрузка — распространённый в других ЯП приём для типизации функций, которые
  - Могут принимать разные типы аргументов
  - Тип возвращаемых данных зависит от типов аргументов

Перегрузка функции

```
function add(...args: number[]): number; // сигнатура для чисел
function add(...args: string[]): string; // сигнатура для строк

// реализация функции – максимально широкий тип аргументов и
// возвращаемого значения
function add(...args: number[] | string[]): number | string {
  if (isArrOfNumbers(args)) {
    return args.reduce((sum, val) => sum + val, 0);
  }

  return args.reduce((sum, value) => sum + value, '');
}

const numResult: number = add(2, 2, 3); // 7
const strResult: string = add('lol', 'kek'); // lolkek
```

*весь код тут

Generics