

The image features the TypeScript logo, which consists of the word "TypeScript" in a blue, sans-serif font. The text is positioned in the upper right quadrant of the image. Below the text, there is a dark blue silhouette of a city skyline with various building shapes. The background is a light gray gradient, and a large, white, stylized cloud shape is positioned behind the text.

TypeScript

Intro

TypeScript — мультипарадигменный язык программирования, надстройка над JavaScript от компании Microsoft, которая была призвана облегчить написание клиентской части приложения и спасти мир от «нетипизированного JS».

Язык был разработан чтобы расширить возможности ECMAScript5 и внедрить планируемые изменения стандарта ECMAScript6.

Преимущества:

- объектно ориентирован
- строго типизирован
- функционален
- обратно совместим с JavaScript

Базовые типы данных

JavaScript

- boolean
- number
- string
- null
- undefined

TypeScript

- boolean
- number
- string
- array
- enum
- any
- void
- ~~➤ tuple~~
- ~~➤ never~~
- null
- undefined

Any

Этот тип – одна из приятных особенностей Typescript. Он используется, чтобы определить своего рода контейнер, который может принимать любые значения.

ОК, тогда в чем разница с Object?

Действительно, для **Object** можно присвоить любое значение, но при попытке вызвать несуществующий метод или свойство – **компилятор будет не доволен**.

При использовании **Any**, ошибки не будет, потому что компилятор понимает что метод\свойство может появиться в этом типе во время исполнения.

```
31     private anyVariable: any;
32
33     private testAny() {
34         this.anyVariable.isInfopulseUniverCool();
35     }
36
```

AppComponent

Terminal

+ (c) 2017 Microsoft Corporation. All rights reserved.

×

C:\Users\Vladyslav\IdeaProjects\test-project>ng serve

** NG Live Development Server is running on http://localhost:4200 **

Hash: f1379a8a471bd09585f9

Time: 8321ms

chunk {0} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 183 kB {4} [initial] [rendered]

chunk {1} main.bundle.js, main.bundle.js.map (main) 5.13 kB {3} [initial] [rendered]

chunk {2} styles.bundle.js, styles.bundle.js.map (styles) 9.77 kB {4} [initial] [rendered]

chunk {3} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.48 MB [initial] [rendered]

chunk {4} inline.bundle.js, inline.bundle.js.map (inline) 0 bytes [entry] [rendered]

webpack: Compiled successfully.

Boolean

Базовый тип данных.

```
private isTypeDefined: boolean = true; // true
private isDefaultValueDefined: boolean; // false
private isTypePresent = false; // false and type BOOLEAN!
```

Number

Как и в стандарте ECMAScript 2015, в TypeScript все числа – имеют значение с плавающей запятой

```
private firstNumberExample = 7;
private secondNumberExample: number = 7;

private firstFloatExample: number = 0.2323;
private secondFloatExample = 9.3434;
```

String

Еще один фундаментальный тип – это **string**. Используется для определения текстового типа. По сколько как и в JS, у Typescript нет типа «символ», то строки можно объявлять разными способами.

```
private stringSingleQuote = 'my first string!';  
private stringSingleQuoteWithType: string = 'my second string!';  
  
private stringDoubleQuote = "my third string!";  
private stringDoubleQuoteWithType: string = 'my fourth string!';  
  
private stringSingleQuote = 'my fifth string';  
private stringDoubleQuote = "my sixth string";
```

Массивы

Существует несколько способов объявления массивов.

```
private array = [1, 2, 3, 4, 5];
private stringArray: number[] = [1, 2, 3, 4, 5];
private arrayType: Array<string> = ['1', '2', '3', '4'];

private forEachExamples() {

    for (let i = 0; i < this.array.length; i++) {
        console.log(this.array[i]);
    }

    this.stringArray.forEach( callbackfn: element => {
        console.log(element);
    });
}
```

Tuple

Специфический и редко используемый тип. Предоставляет возможность объявлять массив с элементами разных типов данных

```
private myTuple: [number, string, boolean] = [1, '2', true];
```

Enum

Этот тип позволяет задать более дружелюбные имена для используемых значений. С версии typescript 2.4 + появилась возможность добавления enum со **String** значением. **Enum** позволяет нам определить набор констант и удобно переиспользовать их в нашем коде.

```
export enum EnumExample {  
    First = 1,  
    Second = 2,  
    Third = 3  
}
```

```
private useEnum() {  
    console.log(EnumExample.First);  
}
```


Void

Обратно пропорционален типу **any**. Означает полное отсутствие возвращаемого типа. Применяется, когда функция не возвращает значения.

```
private useEnum(): void {  
    console.log(EnumExample.First);  
}
```

Null и Undefined

- Имеют схожее поведение с null & undefined в ECMAScript5.
- Null – значение неизвестно
- Undefined – значение не присвоено.
- Они равны друг другу и не равны больше ничему. При преобразовании в число, null – 0, undefined - NaN

Type Assertions

Это способ сказать компилятору, что Вы знаете о типе переменной больше чем сам **typescript**.

Если кто-то подумал о «**cast**», Вы абсолютно правы, но здесь компилятор не выполняет дополнительной реструктуризации данных.

Существует два способа каста:

➤ angle-bracket

```
this.stringLengthCounter = (<string> this.iAmStupidLibrary()).length;
```

➤ as syntax

```
this.stringLengthCounter = (this.iAmStupidLibrary() as string).length;
```

PS: stringLengthCounner: number, iAmStupidLibrary(): any

Type Assertions

```
private stringLengthCounter: number;
```

```
private iAmStupidLibrary(): any {  
    return 'I am the string with any type! :(';  
}
```

```
private countStringLength() {  
    this.stringLengthCounter = this.iAmStupidLibrary().length;  
    console.log(this.stringLengthCounter);  
}
```

```
ngAfterViewInit() {  
    this.countStringLength();  
}
```

Ошибка будет?

Type Assertions

```
export class AppComponent implements OnInit {  
  
    private stringLengthCounter: number;  
  
    private testFunction(): any {  
        return 1234567;  
    }  
  
    private countStringLength() {  
        this.stringLengthCounter = (this.testFunction() as string).length;  
        console.log(this.stringLengthCounter);  
    }  
  
    ngOnInit() {  
        this.countStringLength();  
    }  
}
```

← Какое значение?
undefined!

Каст не выполнен, а у типа Number нет свойства length.

Не стоит забывать что в **typescript** как и в других языках, **существуют ограничения на прямое преобразование типов** в связи с участками памяти, которые типы резервируют под себя.

https://www.w3schools.com/js/js_type_conversion.asp

Переменные внутри декоратора class

Существует несколько вариантов работы с переменными.

➤ работа с переменными внутри декоратора typescript – class

Эти переменные имеют идентификаторы доступа как и в классических языках программирования

- private `private myPrivateVariable;`
- public `public myPublicVariable;`
- protected `protected myProtectedVariable;`

По мимо этого есть возможность огласить неизменяемую переменную readonly. **Важный** момент, что инициализировать её можно как при объявлении, так и в конструкторе.

```
readonly myInitializedReadOnlyVariable = 'Say hello!';  
readonly myNotInitializedReadOnlyVariable;  
  
constructor () {  
    this.myNotInitializedReadOnlyVariable = 'Say goodbye!';  
}
```

Переменные внутри вне декоратора class или внутри функции

Работа с переменными вне класса или внутри класса частично соответствует стандарту **ECMAScript2015**.

Здесь важно понимание **scope** переменной.

Var

Относятся ко всему scope функции

```
public outerFunction() {  
  
    var a = 5;  
  
    function innerFunction() {  
        console.log(a);  
    }  
  
    innerFunction(); // выведет 5  
}
```

```
public outerFunction(isTrue: boolean) {  
  
    if (isTrue) {  
        var a = 5;  
    }  
  
    console.log(a); // выведет undefined  
}  
  
ngAfterViewInit() {  
    this.outerFunction(isTrue: false);  
}
```

Задание

```
export class AppComponent implements AfterViewInit {  
  title = 'app works !!!!!';  
  
  public outerFunction() {  
    var a = 5;  
  
    function innerFunction() {  
      var a = 6;  
  
      function deepInnerFunction() {  
        console.log(a);  
        var testVariable = 4;  
      }  
  
      deepInnerFunction(); // (1)  
      console.log(testVariable);  
    }  
  
    innerFunction(); // (2)  
    console.log(testVariable); // (3)  
  }  
  
  ngAfterViewInit() {  
    this.outerFunction();  
  }  
}
```

Переменная let

Этот тип переменных относится к **block-scoping**. Это значит что их область видимости по сути ограничивается скобками.

```
private outerFunction(isTrue: boolean) {  
  let outer = 5;  
  
  if (isTrue) {  
    console.log(outer);  
    let inner = 10;  
  }  
  
  console.log(inner); // can't find name inner  
}  
  
ngAfterViewInit() {  
  this.outerFunction(isTrue: true);  
}
```

Readonly

Отличным выбором для создания финализированной переменной будет использование оператора **readonly**. Он добавляет иногда очень полезную возможность инициализировать себя с **конструктора** (по желанию).

```
readonly USER_CONTACT_URL = 'client-base/user/contact/';  
readonly USER_GET_URL;  
  
constructor() {  
  this.USER_GET_URL = '/app/get/user/';  
}
```


Interfaces

Один из самых мощных и удобных инструментов, доступных в **typescript**. Как и в других языках программирования позволяет выносить «**шаблоны**» необходимых структур для последующего использования.

```
interface CarInterface {  
  
    brand: string;  
    type: string;  
    maxSpeed: number;  
    price: number;  
    engineType?: any;  
  
    getCarBrand(): string;  
  
    showMaxSpeed(): number;  
  
    introduceCar(): string;  
}
```

```
...  
export class AppComponent implements CarInterface {  
  
    brand: string;  
    type: string;  
    maxSpeed: number;  
    price: number;  
  
    getCarBrand(): string {  
        return undefined; // your implementation  
    }  
  
    showMaxSpeed(): number {  
        return undefined; // your implementation  
    }  
  
    introduceCar(): string {  
        return undefined; // your implementation  
    }  
}
```

Classes

Одно из важнейших преимуществ **TypeScript** – возможность создавать и работать с **классами**. Классом можно полностью можно описать сущность реального мира.

```
export class Vehicle {

    private _brand: string;
    private _power: number;
    private _maxSpeed: number;
    private _color: string;

    constructor(brand: string, power?: number, maxSpeed?: number,) {
        this._brand = brand;
    }

    public introduceVehicle() {
        return 'It`s a ' + this._brand + ' top vehicle!';
    }

    get maxSpeed(): number {
        return this._maxSpeed;
    }

    set maxSpeed(value: number) {
        this._maxSpeed = value;
    }

    get brand(): string {
        return this._brand;
    }

    set brand(value: string) {
        this._brand = value;
    }
}
```

Наследование

```
export class Car extends Vehicle {
```

```
  private _tyresType;  
  private _transmissionName;  
  private _fuelRate;
```

```
  constructor(brand: string, tyrestype?, fuelRate?) {  
    super(brand);  
    this._tyresType = tyrestype;  
    this._fuelRate = fuelRate;  
  }
```

```
  get tyresType() {  
    return this._tyresType;  
  }
```

```
  set tyresType(value) {  
    this._tyresType = value;  
  }
```

```
export class AppComponent {
```

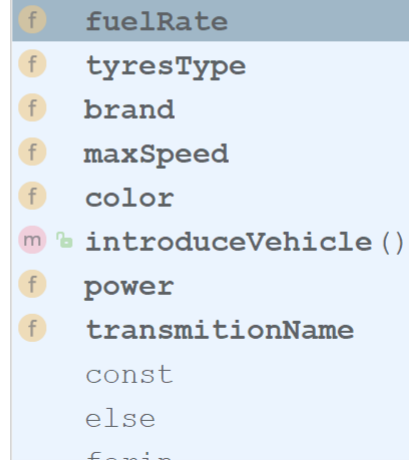
```
  private mercedesBenz = new Car('mercedes-benz');
```

```
  private carSample() {
```

```
    this.mercedesBenz.
```

```
  }
```

```
}
```



Autocomplete dropdown menu showing properties and methods of the Car class:

- fuelRate
- tyresType
- brand
- maxSpeed
- color
- introduceVehicle()
- power
- transmissionName
- const
- else
- finally

Ctrl+Down and Ctrl+Up will move caret down a

Наследование полный пример

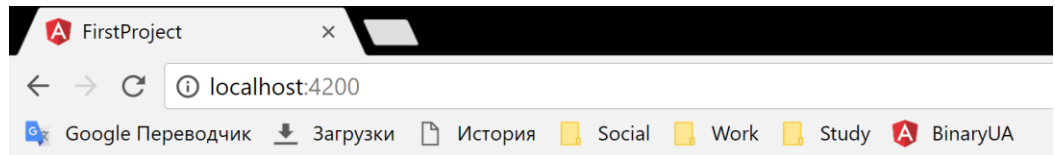
```
@Component ({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements AfterViewInit {

  private carIntroduction;

  private mercedesBenz = new Car('mercedes-benz');

  private carSample() {
    this.mercedesBenz.maxSpeed = 320;
    this.carIntroduction = this.mercedesBenz.introduceVehicle();
  }

  ngAfterViewInit() {
    this.carSample();
  }
}
```



It's a mercedes-benz top vehicle!

Предложите улучшения!

Abstract classes

Абстрактные классы полезны, если неизвестна реализация одного из механизмов системы. Важно помнить, что **нельзя** создать экземпляр абстрактного класса.

```
export abstract class Template {  
  
  readonly COLOR = '#ff8a55';  
  private _width;  
  private _height;  
  
  abstract paintVehicle();  
  
  private calculateSquare() {  
    return this._width * this._height;  
  }  
  
  get width() {  
    return this._width;  
  }  
  
  set width(value) {  
    this._width = value;  
  }  
  
  get height() {  
    return this._height;  
  }  
  
  set height(value) {  
    this._height = value;  
  }  
}
```

```
import {Template} from './abstract.model';  
  
export class PaintingThingModel extends Template {  
  
  // your unic variables and methods  
  
  paintVehicle() {  
    //realization  
  }  
}
```

Вопрос: можно ли создать конструктор внутри абстрактного класса?

Generics

Позволяет убедиться, что все свойства работают с одним и тем же типом. В **typescript** существует возможность создавать методы, интерфейсы и классы параметризованные дженериком.

```
interface MainEntity<T> {  
  id: number;  
  value: T;  
  
  workWithValue<T1>(value: T1);  
}
```

```
class MainEntity<T> {  
  id: number;  
  value: T;  
  
  workWithValue<T1>(value: T1) {  
    // implemented logic  
  }  
}
```

```
export class GenericMain implements MainEntity<string> {  
  id: number;  
  value: string;  
  
  workWithValue<T1>(value: T1) {  
    return typeof value;  
  }  
}
```

```
export class GenericMain extends MainEntity<string> {  
  
  private useSuperClass() {  
    this.workWithValue<number>(value: 56);  
    this.workWithValue<string>(value: 'Hi, am string!');  
  }  
}
```

Стандартные структуры для работы с данными

Оператор сравнения if/else

Оператор условия **if/else**, один из наиболее используемых в программировании.

Служит для проверки определенного условия и выполнения соответствующего действия в зависимости от результата проверки.

```
private showOperatorSample(condition: boolean) {  
  
  if (condition) {  
    // logic if condition is true  
  } else {  
    // logic if condition is false  
  }  
  
  if (!condition) { // if condition is false, stop checking, else continue  
    // business logic  
  } else if (condition == null) { // if condition is null, stop checking, else continue  
    // business logic  
  } else { // if no one statement match  
    // business logic  
  }  
  
  (condition ? console.log(true) : console.log(false)); // ternary operator  
}
```

Операторы перебора элементов

Используются, когда нужно перебрать все элементы заданной последовательности.

```
private showOperatorSample(array: number[]) {  
  
    for (let i = 0; i < array.length; i++) {  
        console.log(array[i]);  
    }  
  
    for (let elementArray of array) {  
        console.log(elementArray);  
    }  
  
    array.forEach( callbackfn: (arrayElement: number) => {  
        console.log(arrayElement);  
    });  
}  
  
ngAfterViewInit() {  
    var userArray = [1, 2, 3, 4, 5];  
    this.showOperatorSample(userArray);  
}
```


Конструкция switch/case

Представляет собой еще один вариант сравнения выражения с несколькими вариантами.

```
private showOperatorSample(color: Color) {  
  
    switch (color) {  
        case Color.BLACK:  
            console.log('It`s black!');  
            break;  
        case Color.GREEN:  
            console.log('It`s green!');  
            break;  
        case (Color.YELLOW):  
            console.log('It`s yellow!')  
            break;  
        default:  
            console.log('Unknown color!');  
            break;  
    }  
}  
  
ngAfterViewInit() {  
    this.showOperatorSample(Color.GREEN);  
}
```

Цикл while-true

Один из способов перебора, который гарантирует продолжение выполнения, пока не будет выполнено определенное условие.

```
private showOperatorSample() {  
  
    var i = 0;  
  
    while (i < 10) {  
        console.log(i);  
        i++;  
    }  
  
    while (true) {  
        // will run forever  
    }  
}
```

Цикл do-while

Способ перебора, при котором точно выполнится одна итерация цикла.

```
private showOperatorSample() {  
  
    var a = 0;  
    do { // loop will run for 10 times  
        console.log(a);  
        a++;  
    } while (a < 10);  
  
    var b = 0;  
    do { // loop will run forever  
        console.log(b);  
    } while (b < 10);  
  
    var c = 10;  
    do { //loop will run 1 time  
        console.log(c);  
        c++;  
    } while (c < 10);  
  
}
```