Лекция 21. TypeScript

[**Обзор TypeScript**](#_llonzkx8xmgp) **3**

[Установка и компиляция из командной строки](#_kxagz5mcc0qz) 3

[Настройки компиляции](#_akrstueo6qxz) 3

[Файл конфигурации tsconfig.json](#_b4aicr8fnt5o) 4

[**Особенности TypeScript**](#_2g86jlkdr5yh) **5**

[Переменные и константы](#_o819rrrfo9y1) 5

[Типы данных](#_e853x5eodefd) 5

[Базовые типы данных](#_63zdfzy0z2p8) 5

[Объявление массивов](#_6kqn1qr02r6w) 6

[Кортеж](#_nhvkbtuxw7fi) 6

[Тип enum](#_ozy09ywda6q9) 7

[Работа с типами данных](#_u2j5jr7c2fcr) 7

[Объединения](#_xl0orj4dfb1) 7

[Проверка типа](#_9th0l5z512lb) 7

[Псевдонимы типов](#_89imvdyhcxss) 8

[Тип assertion](#_fzbyj9ibowqe) 8

[Функции](#_d3ga8jrrlr6i) 9

[Функции обратного вызова](#_7gvecmkijf2b) 9

[**ООП в TypeScript**](#_q80g9h53ajjn) **9**

[Классы](#_p96t5rj7h7tk) 9

[Определение атрибутов вне конструктора:](#_g9dorrtj8cr7) 9

[Статичные атрибуты и методы:](#_lcoleczguffp) 10

[Модификаторы доступа](#_j3ptk57parkz) 10

[Private](#_2cmw8tw1miy7) 10

[Protected](#_mk0986bspdrr) 11

[Public](#_qsa1ipiiocj8) 12

[Методы доступа и readonly-свойства](#_ot2922qtxvgb) 12

[Наследование. Абстрактные классы](#_9cql15rnxdv2) 12

[Интерфейсы](#_wsmh5c5r7oqi) 14

[Применение абстрактных классов и интерфейсов](#_sz6rzggd3mrs) 15

[Преобразование типов](#_q65ga7dxhffq) 17

[**Дополнительные материалы**](#_aljeicoq30c3) **18**

[**Задание для закрепления материала**](#_y4fwt8akcgv6) **18**

# Обзор TypeScript

Разработан и поддерживается Microsoft. Впервые был представлен в октябре 2012.

В общем и целом, расширяет возможности es6 до языков более высокого уровня. Добавляются новые типы (ENUM), строгая типизация, интерфейсы и тд.

## Установка и компиляция из командной строки

Установить TypeScript можно командой:  
 **npm install -g typescript**

Для проверки версии необходимо ввести команду

**tsc -v**

Компиляция

**tsc app.ts**

**tsc {filename}**

## Настройки компиляции

**Автоматическая перекомпиляция**

Опция --watch, а также ее сокращенная версия -w автоматически перекомпилирует файлы typescript, если в них были внесены какие-либо изменения. Благодаря чему не надо при каждом малейшем изменении вручную вводить команду в консоль для перекомпиляции.

tsc -w app.ts

**Версия ECMAScript**

С помощью параметра –-target или его сокращенной версии –t можно задать версию стандарта JavaScript, в которую будет компилироваться код TypeScript. Этот параметр может принимать следующие значения: "ES3" (по умолчанию), "ES5", "ES6", "ES2015", "ES2016", "ES2017" или "ESNext":

tsc app.ts -t ES5

**Удаление комментариев**

По умолчанию в файлы javascript переходят все комментариии, которыми снабжен код в файлах TS. Удаление комментариев при компиляции осуществляется с помощью параметра –-removeComments:

tsc app.ts --removeComments

**Установка каталога**

С помощью параметра --outDir можно задать папку для хранения скомпилированных файлов js:

tsc --outDir D:\ts\js app.ts

В данном случае скомпилированный файл app.js окажется в папке D:\ts\js

**Объединение файлов**

Если у нас несколько файлов TS, то с помощью параметра --outFile их можно объединить в один файл js:

tsc --outFile output.js app.ts hello.ts

Здесь файлы app.ts и hello.ts скомпилируются в один файл output.js

**Тип модуля**

С помощью параметра --module, либо -m можно указать тип модуля, который будет использоваться для компиляции. Эта опция может принимать следующие значения: "None", "CommonJS", "AMD", "System", "UMD", "ES6", "ES2015" и "ESNext".

## Файл конфигурации tsconfig.json

С помощью файла tsconfig.json можно настроить проект TypeScript. В частности, этот файл выполняет следующие задачи:

устанавливает корневой каталог проекта TypeScript

выполняет настройку параметров компиляции

устанавливает файлы проекта

Для его использования достаточно вручную добавить новый файл с именем tsconfig.json в корень проекта.

структуру и примеры можно посмотреть на :  
<https://metanit.com/web/typescript/1.5.php>

# Особенности TypeScript

## Переменные и константы

Использование let и const осталось неизменным. В любом случае, let и const предпочтительнее, чем var.

## Типы данных

### Базовые типы данных

В TypeScript имеются следующие базовые типы:

**Boolean**: логическое значение true или false

**Number**: числовое значение

**String**: строки

**Array**: массивы

**Tuple**: кортежи

**Enum**: перечисления

**Any**: произвольный тип

**Null и undefined**: соответствуют значениям null и undefined в javascript

**Void**: отсутствие конкретного значения, используется в основном в качестве возвращаемого типа функций

**Never**: также представляет отсутствие значения и используется в качестве возвращаемого типа функций, которые генерируют или возвращают ошибку

**let x: number = 10;**

**let hello: string = "hello world";**

**let isValid: boolean = true;**

в данном примере мы объявили 3 переменных разного типа: number, string, boolean.

Основное отличие от es6 это то, что мы больше не сможем поместить в эти переменные данные, несоответствующие по типу, т.е.

**x = ‘test string’;**  
  
уже вызовет ошибку компиляции.

переопределить переменную можно будет только выйдя за пределы блочной видимости.

При этом, даже если не указывать тип явно:

**let hello = "hello world";**

**hello = 23;**

компилятор все равно выведет ошибку, т.к. тип переменной определяется автоматически при инициализации, т.е. string.

Исключением является тип any:

**let x: any;**

**x = ‘test string’;**

**x = 51;**

Также, в случае если значение не определено при объявлении:

**let x;**

**x = ‘test string’;**

**x = 51;**

тип будет определен как any.

### Объявление массивов

Элементам массива также можно задать тип:

**let list: number[] = [10, 20, 30];**

**let colors: string[] = ["red", "green", "blue"];**

**console.log(list[0]);**

**console.log(colors[1]);**

или альтернативным способом:

**let names: Array<string> = ["Tom", "Bob", "Alice"];**

### Кортеж

Кортеж - это массив, в котором мы можем определить тип для каждого элемента:

**let userInfo: [string, number];**

**// инициализация кортежа**

**userInfo = ["Tom", 28];**

### Тип enum

Тип enum предназначен для описания набора числовых данных с помощью строковых констант. Так, объявим следующее перечисление:

**enum Season { Winter, Spring, Summer, Autumn };**

**let current: Season = Season.Summer;**

**console.log(current);**

**current = Season.Autumn; // изменение значения**

Также можно определить числовые значения для каждого элемента:

**enum Season { Winter=0, Spring=1, Summer=2, Autumn=3 };**

**var current: string = Season[2]; // 2 - числовое значение Summer**

**console.log(current); // Summer**

## Работа с типами данных

### Объединения

Объединения или union не являются собственно типом данных, но они позволяют определить переменную, которая может хранить значение двух или более типов:

**let id : number | string;**

**id = "1345dgg5";**

**console.log(id); // 1345dgg5**

**id = 234;**

**console.log(id); // 234**

### Проверка типа

**let sum: any;**

**sum = 1200;**

**sum = "тысяча двести";**

**let result: number = sum / 12;**

**console.log(result); // NaN - строку нельзя разделить на число**

**// попробуем иначе**

**let sum: any;**

**sum = 1200;**

**if (typeof sum === "number") {**

**let result: number = sum / 12;**

**console.log(result);**

**}**

**else{**

**console.log("invalid operation");**

**}**

### Псевдонимы типов

TypeScript позволяет определять псевдонимы типов с помощью ключевого слова type:

**type stringOrNumberType = number | string;**

**let sum: stringOrNumberType = 36.6;**

**if (typeof sum === "number") {**

**console.log(sum / 6);**

**}**

### Тип assertion

**let someAnyValue: any = "hello world!";**

**let strLength: number = (<string>someAnyValue).length;**

**console.log(strLength); // 12**

**let someUnionValue: string | number = "hello work";**

**strLength = (<string>someUnionValue).length;**

**console.log(strLength); // 10**

или

**let someAnyValue: any = "hello world!";**

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

**console.log(strLength); // 12**

**let someUnionValue: string | number = "hello work";**

**strLength = (someUnionValue as string).length;**

**console.log(strLength); // 10**

## Функции

Основным новшеством TypeScript по сравнению с ES6, является то, что для параметров функции можно указать тип:

**function sum (x: number, y: number) {**

**return x + y;**

**};**

а также тип возвращаемого значения:

**function sum (x: number, y: number): number {**

**return x + y;**

**};**

### Функции обратного вызова

**function mathOp(x: number, y: number, operation: (a: number, b: number) => number): number {**

**let result = operation(x, y);**

**return result;**

**}**

Т.е. в качестве параметра для функции мы также можем указать описание функции с типами параметров и возвращаемого значения.

# ООП в TypeScript

## Классы

Классы в TypeScript, в целом, очень похожи на классы в ES6. Можно выделить несколько отличий:

### Определение атрибутов вне конструктора:

**class User {**

**id: number;**

**name: string;**

**constructor() {}**

**getInfo(): string {**

**return "id:" + this.id + " name:" + this.name;**

**}**

**}**

### Статичные атрибуты и методы:

К статичным атрибутам и методам можно обращаться непосредственно из класса, не объявляя объект:

**class Operation {**

**static PI: number = 3.14;**

**static getSquare(radius: number): number {**

**return Operation.PI \* radius \* radius;**

**}**

**}**

**let result = Operation.getSquare(30);**

**console.log("Площадь круга с радиусом 30 равна " + result);**

**let result2 = Operation.PI \* 30 \* 30;**

**console.log(result2); // 2826**

## Модификаторы доступа

Одной из ключевых концепций объектно-ориентированного программирования является инкапсуляция, подразумевающая сокрытие от внешнего доступа к состоянию объекта и управление доступом к этому состоянию. Обычно для этого во многих ООП-языках (как Java, C#) используются модификаторы доступа. В TypeScript три модификатора: public, protected и private.

### Private

Атрибуты и методы, объявленные как private будут доступны только для объектов данного класса (даже потомкам этого класса private методы доступны не будут)

**class User {**

**private name: string;**

**private year: number;**

**constructor(name: string, year: number) {  
 this.name = name;  
 this.year = year;  
 }**

**}  
  
let Dan = new User(“Dan”, 1984);  
console.log(Dan.name, Dan.year); // ошибка доступа**

**Dan.name = “Emily”; // ошибка доступа**  
  
Для чего это нужно? Один из принципов ООП - инкапсуляция обязывает нас скрывать детали реализации. Нет возможности обратиться к атрибутам напрямую, - нет и искушения нарушить один из основных принципов ООП.

Как можно обратиться к атрибутам, если они спрятаны? Для этого используются т.н. геттеры и сеттеры:

**class User {**

**private name: string;**

**private year: number;**

**constructor(name: string, year: number) {  
 this.name = name;  
 this.year = year;  
 }**

**public function getName() {  
 return this.name;  
 }  
 public function setName(name: string){  
 this.name = name;  
 }**

**}**

**let Dan = new User(“Dan”, 1984);**

**console.log( Dan.getName() ); // Dan**

Стоит отметить, что private методы/атрибуты не наследуются дочерними классами.

### Protected

Единственным отличием protected методов от private является то, что protected наследуется потомками.

### Public

Отличие public методов от остальных модификаторов - это доступность таких методов и атрибутов отовсюду, и извне класса и из потомков класса. Стоит отметить, что если не применять модификаторы доступа к атрибутам и методам, то они , по-умолчанию, будут public.

## Методы доступа и readonly-свойства

Как уже упоминалось ранее, для доступа к закрытым членам класса используются геттеры и сеттеры.

Readonly атрибуты отличаются тем, что установить их можно только из конструктора класса:

**class User {**

**readonly id: number;**

**name: string;**

**constructor(userId: number, userName: string) {**

**this.id = userId;**

**this.name = userName;**

**}**

**}**

**let tom: User = new User(2, "Tom");**

**console.log(tom.id, tom.name);**

**//tom.id=34; // Ошибка - так как id - только для чтения**

## Наследование. Абстрактные классы

Абстрактные классы служат для отражения и описания групп объектов, как таковой, объект группы существовать не может, существовать может конкретная реализация, например:  
абстрактная геометрическая фигура, существовать не может

а вот конкретная фигура, например: круг, квадрат, прямоугольник; вполне.

Очень хорошо представить абстрактные классы можно на примере каталога в интернет магазине:

**Бытовая техника**

**Крупная бытовая техника  
 Встраиваемая техника  
 Холодильники  
 Indesit A5500   
 Bosch B1801**  
 …

У классов товара “Бытовая техника”, “Встраиваемая техника” и тд. не может быть собственных характеристик, т.к. тогда эти характеристики распространились бы на все вложенные классы/объекты. Мы лишь можем утверждать, что у всех объектов этих групп есть такие параметры как ширина/высота/глубина, конкретных же цифр мы назвать не можем, это слишком разнится от элемента к элементу

А вот у конкретной реализации : Indesit A5500 или Bosch B1801 эти параметры вполне конкретные.

В этом и заключается идея Абстрактного класса: мы можем сказать что какие-либо параметры/методы (холодильники морозят, поддерживают температуру) у экземпляра класса имеются и должны быть определены/реализованы. Как реализованы - это дело дочернего класса или непосредственно объекта (инкапсуляция). Но исходя из текущего уровня мы можем гарантировать, что все объекты будут иметь реализацию какого-либо метода и иметь какие-то атрибуты.

**Абстрактный метод в абстрактном классе говорит о том, что этот метод обязательно должен быть реализован в классе-потомке. Если у класса есть хоть один абстрактный метод, класс должен быть объявлен абстрактным!**

**// бытовая техника**

**abstract class Appliances {  
 public height: number;**

**public width: number;  
 public depth: number;  
  
 abstract getWareHouseAmount(): number;  
}**

**…**

**// холодильники**

**abstract class Freezer extends Appliances {  
 abstract freeze(): void;**

**abstract keepTemperature(): void;**

**}  
…  
class IndesitA5500 extends Freezer {**

**// тут конкретная реализация**

**public height: number = 500;  
 public width: number = 500;  
 public depth: number = 500;  
 freeze() {  
 // реализация метода “морозить”  
 }  
 keepTemperature() {  
 // реализация метода “поддерживать температуру”  
 }  
 getWareHouseAmount() {  
 // реализация метода getWareHouseAmount  
 }  
}**

В примере выше методы freeze() и keepTemperature() должны быть реализованы, т.к. описаны в абстрактном классе Freezer.

Помимо абстрактных методов абстрактный класс может содержать привычные нам методы, которые будут унаследованы потомками, и при этом содержать какую-либо реализацию внутри себя.

## Интерфейсы

Интерфейсы похожи на абстрактные классы тем, что обязуют классы реализовывать методы, но архитектурно предназначены совершенно для другого: они лишь гарантируют что класс/объект имеют свою реализацию метода, интерфейс реализовывать могут объекты совсем друг с другом не связанные (т.е. вообще никак) и не имеющие общего родителя. Например:  
нам необходимо реализовать возможность сесть на объект (стул/кресло/...)  
  
**class Chair {  
 public seatOnIt() {  
 ...  
 }  
}**

Стоп! но и на стол мы можем сесть (не рекомендуется, но практикуется). Окей, добавим родителя:

**abstract class Furniture {  
 // абстрактный класс “мебель”  
 abstract seatOnIt():void;  
}**

**class Chair extends Furniture {  
 public seatOnIt() {  
 ...  
 }  
}**

**class Table extends Furniture{  
 public seatOnIt() {  
 ...  
 }  
}**

Хорошо, мы уже сидим на столе, но мы также можем сидеть на подоконнике или камне или стволе дерева! К мебели это отношения никакого не имеет, именно в этом месте нам очень полезны интерфейсы:  
  
**interface SeatableInterface {  
 public seatOnIt():void;  
}**

**class Chair implements SeatableInterface {  
 public seatOnIt() {  
 ...  
 }  
}**

**class Table implements SeatableInterface {  
 public seatOnIt():void {  
 ...  
 }  
}**

**class Stone implements SeatableInterface {  
 public seatOnIt():void {  
 ...  
 }  
}**

Теперь мы можем сесть и на Кресло и на Стол и на Камень. Притом эти объекты не связаны общим предком, и мы о них знаем, что на них точно можно сесть. Кстати, реализация метода seatOnIt() может быть разной в каждом случае - это и есть полиморфизм.

Дополнительным отличием интерфейсов от абстрактных классов является то, что интерфейсы не могут содержать реализацию.

## Применение абстрактных классов и интерфейсов

Предположим, что у нас есть класс человек

**class Person {  
 // тут неважные в нашей ситуации атрибуты и методы**

**...**

**// человек может присесть на какой-либо объект**

**public seatOn(ObjectToSeat){**

**...  
 }  
}**

Как определить что объект ObjectToSeat вообще пригоден для того, чтобы на нем сидеть? Это может быть: вилка, ёж или горячая плита. Нет, теоретически, человека можно посадить на все эти предметы в качестве шутки или глупого эксперимента, но на живом проекте для заказчика этого делать категорически нельзя.

Мы можем задать возможность человеку садиться только на мебель:

**class Person {  
 // тут неважные в нашей ситуации атрибуты и методы**

**...**

**// человек может присесть на какой-либо объект**

**public seatOn(ObjectToSeat: Furniture){**

**...  
 }  
}**

в этом случае, ObjectToSeat должен быть мебелью, что тоже не совсем правильно: как мы выяснили - сесть можно не только на мебель, да и не вся мебель пригодна в качестве сиденья. Поэтому, мы и реализовали Интерфейс SeatableInterface и теперь можем его применить:

**class Person {  
 // тут неважные в нашей ситуации атрибуты и методы**

**...**

**// человек может присесть на какой-либо объект**

**public seatOn(ObjectToSeat: SeatableInterface){**

**...  
 }  
}**

Этим мы гарантировали, что у **ObjectToSeat** есть метод **seatOnIt** и он пригоден в качестве сиденья. В случае, если человек пытается присесть на объект, для этого не предназначенный, приложение закроется с ошибкой.

## Преобразование типов

В TypeScript имеется возможность явного преобразования типов:

**class Furniture {  
 public material: string;  
  
}**

**class Chair extends Furniture {**

**public upholstery: string;  
}**

**let myChair: Furniture = new Chair();**

**// это сработает, т.к. Chair явно расширяет Furniture;**

**console.log( myChair.upholstery );**

**// а вот это уже не сработает, т.к. у Furniture нет атрибута upholstery, a myChair объявлен как Furniture.**

**console.log( (<Chair>myChair).upholstery );**

**// тут мы явно указываем компилятору к какому классу относить myChair**

**// или**

**console.log( (myChair as Chair).upholstery );**

# Дополнительные материалы

<https://metanit.com/web/typescript/>

<https://www.typescriptlang.org/>

# Задание для закрепления материала

Доработать архитектуру приложения с использованием возможностей TS:

* строгая типизация
* интерфейсы
* расширение классов
* абстрактные классы
* новых типов данных
* etc.