

Typescript

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

<https://create-react-app.dev/docs/adding-typescript/>

Typescript - Javascript на 80%. **Отличия:**

- позволяет типизировать код.

Преимущества Typescript:

- Можем еще на этапе написания кода найти баги
- Фактически документируем код
- Крупные компании используют типизированные языки
- Более понятный и читаемый код

Браузеры не умеют поддерживать typescript. Нужно переводить TS в JS.

установка

`npm install -g typescript` или

`npm install --save typescript @types/node @types/react @types/react-dom @types/jest` - более расширенная установка

Файлы typescript должны иметь расширение **.ts**.

Файлы typescript с разметкой должны иметь расширение **.tsx**

Компиляция:

`tsc название_файла.ts`

Если при отладке кода не хотим пока типизировать, то временно можно добавить:

`// @ts-ignore`

Типы данных в TypeScript

В TypeScript при объявлении переменной мы обязаны указать ее тип. Для этого после имени переменной мы должны написать двоеточие, а затем указать желаемый тип. Примеры:

```
let test: string = 'abc';
let test: number = 123;
let test: boolean = true;
let test: null = null;
let test: undefined = undefined;
```

Не обязательно задавать значение переменной сразу при ее объявлении. Можно сначала объявить переменную, а затем ниже в коде присвоить ей значение. Пример:

```
let test: string;
test = 'abc';
```

Если мы **не укажем тип переменной**, TypeScript сам это отслеживать (типизировать автоматически), но это плохая практика.

TypeScript, в отличие от других языков со строгой типизацией, **позволяет выполнять операции над разными типами данных**, не преобразуя их в один общий тип. То есть можно складывать, например, строки и числа, и это не приведет к ошибке. То есть TypeScript следит только за тем, чтобы программист не изменил тип данных переменной.

```
let test1: number = 123;
let test2: number = 456;
let test3: string = test1 + ' ' + test2;
console.log(test3); //123 456
```

Переменной можно **присваивать несколько типов**, перечисляя их через оператор **|**. В коде ниже переменной назначается два типа: строковый и численный. Теперь переменной можно присваивать как текстовые данные, так и числовые.

```
let multitypeVar: string | number = 'String'
multitypeVar = 20
```

С помощью оператора объединения типов можно объединять не только встроенные типы, но и **строки**. Для примера сделаем так, чтобы переменная могла принимать только одно из двух строковых значений:

```
let str: 'success' | 'error';
str = 'success';
```

```
str = 'error';  
str = 'eee'; // ошибка
```

Иногда нам может понадобиться описать тип переменных, который может быть нам не известен на момент, когда мы пишем приложение. Для этого используется тип **any**, позволяющий пройти проверку значений на этапе компиляции.

```
let test: any;  
test = 123;  
test = 'abc';
```

помощью типа any можно объявить **массив, содержащий значения произвольного типа**:

```
let arr: any[] = ['abs', true, 3];
```

Ключевое слово as

Иногда приходится преобразовывать (кастовать) переменную одного типа в другой тип. Особенно часто это случается, когда переменную типа any (или другого произвольного типа) нужно передать в функцию, которая принимает аргумент определённого типа. Чтобы **кастовать** переменную, нужно после оператора as написать тип, в который переводится переменная.

```
let str: any = 'Текстовая переменная'  
let strLength = (str as string).length
```

В этом коде текстовая переменная str кастуется в тип String, а поэтому можно использовать параметр length (это сработает и без кастования, если есть соответствующее разрешение в настройках TSLINT).

Оператор <>

Выполняет абсолютно такую же работу, что и ключевое слово as.

```
let str: any = 'Текстовая переменная'  
let strLength = (<string>str).length
```

Этот код работает идентично предыдущему — разница только синтаксическая.

Массивы в TypeScript

Массивы в TypeScript являются строго типизированными. Это значит, что они могут содержать в себе только данные одного типа. Тип данных массива определяется двумя способами.

Первый способ

Давайте сделаем массив со строками. Для этого после имени переменной укажем тип данных, а после него напишем квадратные скобки в знак того, что у нас массив:

```
let arr: string[] = ['a', 'b', 'c', 'd', 'e'];
```

Давайте выведем какой-нибудь элемент массива:

```
console.log(arr[0]); // выведет 'a'
```

Второй способ

Существует альтернативный способ объявления массива. В нем мы указываем ключевое слово Array, а затем в угловых скобках указывается тип данных. Смотрите пример:

```
let arr: Array<string> = ['a', 'b', 'c', 'd', 'e'];
```

Объекты в TypeScript

Объекты в TypeScript ведут себя особым образом. При объявлении объектов тип данных не указывается. Для примера сделаем объект с юзером, хранящий в себе его имя и возраст:

```
let user = {name: 'john', age: 30};
```

TypeScript контролирует **тип переменной с объектом**, запрещая записывать в нее данные другого типа.

```
user = 'eric'; // ошибка
```

TypeScript также контролирует **структуру объекта**. В момент объявления TypeScript запоминает, что в нашем объекте есть ключи name и age, а затем контролирует, чтобы в переменной хранился объект именно с этими ключами:

```
let user = {name: 'john', age: 30};
user = {name: 'eric'}; // ошибка
user = {name: 'eric', age: 40, salary: 300}; // ошибка
user = {name: 'eric', age: 40}; // работает
```

В момент объявления объекта TypeScript запоминает тип данных всех его элементов, а затем контролирует, чтобы эти типы не изменялись.

```
user.name = 123; // ошибка
```

Циклы в TypeScript

В цикле **for** нужно указывать тип счетчика:

```
for (let i: number = 0; i <= 10; i++) {
    console.log(i);
}
```

А вот в цикле **for-of** тип переменной для элемента не указывается:

```
let arr: number[] = [1, 2, 3, 4, 5];
```

```
for (let elem of arr) {
    console.log(elem);
}
```

То же касается и цикла **for-in** - тип переменной для ключа не указывается:

```
let obj = {a: 1, b: 2, c: 3};
```

```
for (let key in obj) {
    console.log(key);
}
```

Однако:

```
let obj = {a: 1, b: 2, c: 3};
```

```
let key: keyof typeof obj;
```

```
for (key in obj) {
    console.log(obj[key]);
}
```

Кортежи

Иногда у нас может возникнуть необходимость хранить массив значений **различных типов**. TypeScript предоставляет нам такой тип данных, называемый **кортеж**. Кортеж представляет собой **массив**, каждый элемент которого имеет свой жестко заданный тип. Элементы кортежа можно изменять. Попытка записать в элемент кортежа значение другого типа приведет к ошибке

```
let user: [string, number] = ['john', 31];
user[0] = 'eric';
user[0] = 12; // ошибка
```

Можно создавать кортежи **только для чтения**, элементы которого нельзя изменить. Для этого перед типом кортежа указывается ключевое слово **readonly**: Попытка изменить такой кортеж приведет к ошибке:

```
let user: readonly [string, number] = ['john', 31];
user[0] = 'eric'; // ошибка
```

Кортежи могут иметь **необязательные элементы**, для которых можно не предоставлять значение. Чтобы указать, что элемент является необязательным, после типа элемента ставится **вопросительный знак**.

```
let user: [string, number, boolean?];
user = ['john', 31, true]; //user: ", ["john", 31, true]
user = ['john', 31]; //user: ", ["john", 31]
```

Кортеж, подобно массиву, может быть подвергнут **деструктуризации**.

```
let user: [string, number] = ['john', 31];
let [name, age] = user
```

С помощью оператора **rest** в кортеже можно определить набор элементов, состоящий из произвольного количества значений, имеющих один тип. Для примера давайте сделаем кортеж, в котором первым элементом должна быть строка, а затем будет следовать произвольное количество чисел:

```
let tpl: [string, ...number[]];
tpl = ['str', 1, 2, 3, 4, 5];
```

Перечисления в TypeScript

Перечисления - это удобный способ создания **наборов значений**, количество и названия которых известны заранее и не должны изменяться. Для создания перечислений используется ключевое слово **enum**. Давайте создадим перечисление, содержащее название пор года:

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

Данные в перечислениях можно получать по числовым ключам, подобно элементам массива. А можно по названию элемента получить его **ключ**

```
let current: string = Season[0]; // 'Winter'
let current: number = Season.Winter; // 0
```

Каждое перечисление создает **свой собственный тип данных**. Давайте для примера для переменной, хранящей текущий сезон присвоим тип Season:

```
let current: Season;
```

Запишем в нашу переменную номер сезона:

```
let current: Season = Season.Winter;
console.log(current); // 0
```

Можно указать номер сезона вручную:

```
let current: Season = 3;
```

А вот если попытаться записать данные другого типа, например, строку, то будет ошибка:

```
let current: Season = 'str'; // будет ошибка
```

К сожалению, диапазон значений не отслеживается и можно записать номер, отсутствующий в нашем перечислении:

```
let current: Season = 7; // ошибки не будет
```

При проверке через оператор `typeof` наша переменная отдаст числовой тип:

```
let current: Season = 3;
console.log(typeof current); // "number"
```

Подводя итог можно сказать, что такой тип проверяется не сильно строго и от этого его ценность сомнительна.

Указание номеров

Нумерация номеров не обязательно должна быть с нуля. Можно указать ключи в явном виде следующим образом:

```
enum Season { Winter = 1, Spring = 2, Summer = 3, Autumn = 4 };
```

Упрощенные ключи

Не обязательно указывать ключи всем элементам. Достаточно указать его первому элементу и ключи следующих элементов будут увеличиваться по порядку. Пример:

```
enum Season { Winter = 1, Spring, Summer, Autumn };
let current: Season = Season.Summer;
console.log(current); // выведет 3
```

Строковые перечисления

Ключами могут быть не только числа, но и строки:

```
enum Season {  
    Winter = 'Зима',  
    Spring = 'Весна',  
    Summer = 'Лето',  
    Autumn = 'Осень'  
};  
let current: Season = Season.Summer;  
console.log(current); // 'Лето'
```

Псевдонимы типов в TypeScript

В TypeScript можно создавать псевдонимы типов. Это делается с помощью оператора **type**. Для примера давайте зададим еще одно имя для строкового типа данных:

```
type str = string;  
let test: str = 'abc';
```

Применение

Сделаем новый тип данных, используя объединение типов: Объявим переменную с нашим новым типом. Запишем в нее число. Запишем в нее строку:

```
type stumber = string | number;  
let test: stumber;  
test = 123;  
test = 'abc';
```

Для объединения строк можно ввести свой тип. Давайте сделаем это:

```
type message = 'success' | 'error';  
let str: message;  
str = 'success';
```

Встроенные типы объектов в TypeScript

В JavaScript существует много встроенных классов, имеющих свой тип для объектов. К примеру, объекты с датой, с регулярными выражениями, а также DOM элементы.

Дата

Пусть в переменной будет храниться дата, которую мы создадим средствами JavaScript через команду `new Date`. Давайте запишем в эту переменную объект с датой, содержащей текущий момент времени

```
let date: Date = new Date;
```

А теперь запишем объект с датой, содержащей заданный момент времени:

```
let date: Date = new Date(2030, 11, 31);
```

Регулярки

Давайте сделаем переменную, содержащую регулярное выражение:

```
let reg: RegExp = /.+?/;
```

Либо воспользуемся альтернативным способом задания регулярки:

```
let reg: RegExp = new RegExp('.+?');
```

DOM элементы

Для DOM элементов также есть свои типы данных. Давайте посмотрим работу DOM элементами на примере. Пусть у нас есть следующий див:

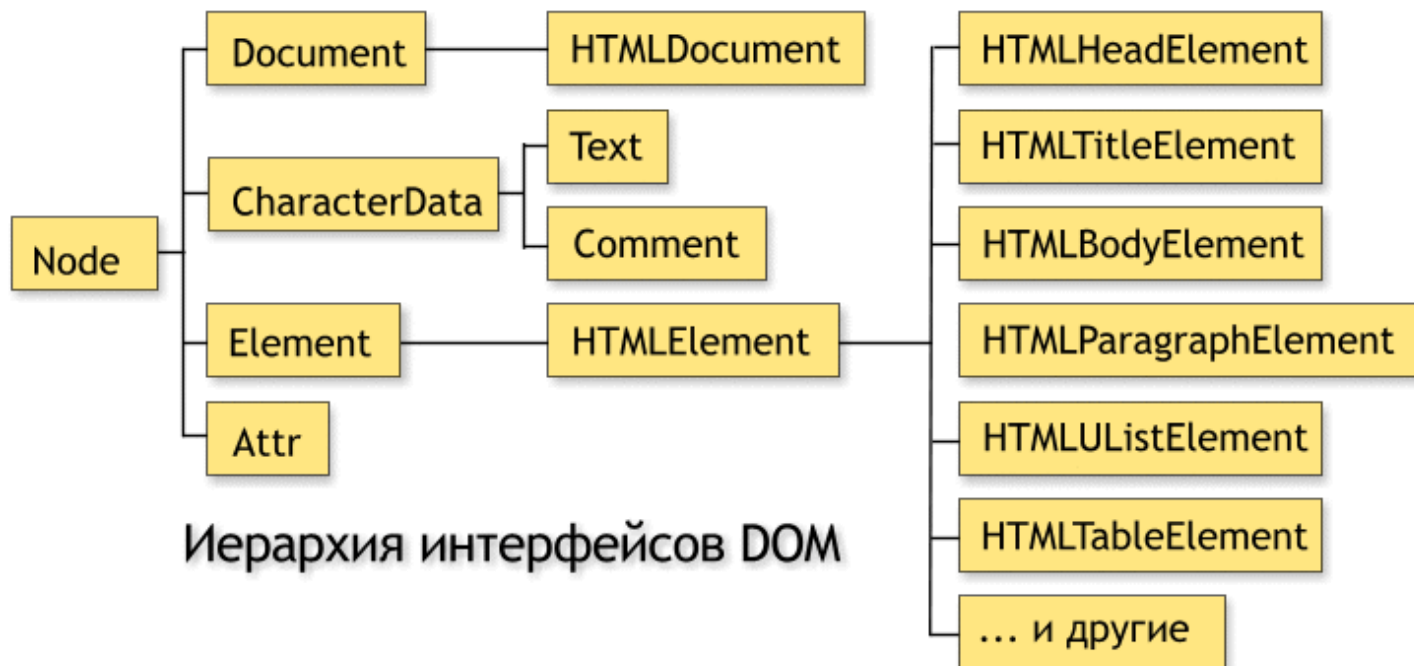
```
<div></div>
```

Давайте получим ссылку на этот див в переменную. Все DOM элементы относятся к типу `HTMLElement`. Укажем этот тип нашей переменной:

```
let elem: HTMLElement = document.querySelector('div');  
console.log(elem);
```

Все дивы, помимо того, что являются DOM элементами с типом `HTMLElement`, также относятся к типу `HTMLDivElement` (аналогичные типы есть и у других тегов). Давайте для нашего элемента укажем более точный тип:

```
let elem: HTMLDivElement = document.querySelector('div');
console.log(elem);
```



Коллекции

Пусть у нас есть несколько дивов:

```
<div></div>
<div></div>
<div></div>
```

Давайте запишем в переменную коллекцию этих DOM элементов:

```
let lst: NodeList = document.querySelectorAll('div');
```

Объект

Можно объявить объект и сразу записать в него значение:

```
let user: {name: string, age: number} = {name: 'john', age: 30};
```

Для разделения свойств, можно использовать как **запятую**, так и **точку с запятой**.

При объявлении объекта можно указать часть свойств как **необязательные**. Для этого после имени свойства нужно указать **знак вопроса**.

```
let user: {name: string, age?: number};
```

Интерфейсы

Более продвинутым вариантом типизации объектов в TypeScript являются **интерфейсы**. Они позволяют создавать новые типы данных, описывающие структуру объектов. Интерфейсы создаются с помощью ключевого слова **interface**, после которого идет название интерфейса (пишется с большой буквы), а затем в фигурных скобках описывается структура объекта.

```
interface User {
  name: string,
  age: number
}
```

Создадим теперь объект, реализующий этот интерфейс. Для этого в качестве типа объекта укажем имя нашего интерфейса:

```
let user: User = {name: 'john', age: 30};
```

Объекты могут иметь внутри себя структуру любой вложенности и эта структура может быть описана при объявлении объекта или в интерфейсе.

```

interface User {
    name: string,
    age: number,
    parents: {
        mother: string,
        father: string
    }
}

let user: User = {
    name: 'john',
    age: 30,
    parents: {
        mother: 'jane',
        father: 'eric'
    }
}

```

Объекты могут содержать в себе другие объекты, описываемые отдельными интерфейсами.

```

interface Employee {
    name: string,
    position: Position,
};

```

```

interface Position {
    name: string,
    salary: number,
};

```

```

let employee: Employee = {
    name: 'andrew',
    position: {
        name: 'programmer',
        salary: 1000,
    },
};

```

Массивы объектов

Массивы в TypeScript могут содержать в себе не только примитивы, но и объекты определенных типов.

```

interface User {
    name: string,
    age: number
}

let arr: User[] = [];
arr.push({name: 'john', age: 30});
arr.push({name: 'eric', age: 40});

```

Функции в TypeScript

При объявлении **параметров функций** мы также можем указать их **тип**. Можно также указать **тип возвращаемого функцией значения**.

```

function func(a: number, b: number): number {
    return a + b;
}

```

Бывают функции, которые **ничего не возвращают**. В этом случае в качестве результата им указывают ключевое слово **void**.

```

function func(test: string): void {
    alert(test);
}

```

!! В TypeScript при вызове в функцию должно передаваться **ровно столько значений, сколько в ней определено параметров**.

Можно сделать так, чтобы функция принимала **переменное число параметров**. Для этого нужно объявить часть параметров (или все) необязательными. Чтобы указать, что параметр является необязательным, после его имени нужно поставить **знак вопроса**.

```
function func(first: string, last?: string) {
    if (last) {
        return first + ' ' + last;
    } else {
        return first;
    }
}
```

Необязательным параметрам можно также назначать **значение по умолчанию**.

```
function func(first: string, last: string = 'snow') {
    return first + ' ' + last;
}
func('john', 'smit'); // вернет 'john smit'
func('john'); // вернет 'john snow'
```

В TypeScript можно также работать с **rest параметрами** функций. Для этого переменную, в которую складываются параметры следует объявить массивом:

```
function func(...rest: number[]): void {
    console.log(rest);
}
func(1, 2, 3); // выведет [1, 2, 3]
```

В JavaScript могут быть переменные, хранящие функции. В этом случае TypeScript позволяет нам указать, что эта переменная имеет тип **"функция"**. Тип функции представляет собой комбинацию типов параметров и типа возвращаемого значения. Эта комбинация называется **сигнатурой функции**. Чтобы указать переменной тип функции, нужно в круглых скобках перечислить параметры и их типы, а после стрелки => указать тип возвращаемого значения.

```
let func: (x: number, y: number) => number;
```

```
func = function(a: number, b: number): number {
    return a + b;
};
```

Иногда удобнее объявить отдельный **тип**, который будет содержать описание параметров и возвращаемого значения функции. Затем можно будет объявлять функции с таким типом.

```
type Func = (x: number, y: number) => number;
```

```
let func1: Func = function(a: number, b: number): number {
    return a + b;
};
```

```
let func2: Func = function(a: number, b: number): number {
    return a * b;
};
```

Для **функций коллбэков** также можно указывать тип параметров и тип возвращаемого значения.

```
function make(num: number, func: (num: number) => number): number {
    return func(num);
}
```

```
make(3, function(num: number): number {
    return num ** 2;
}); // 9
```

```
make(3, function(num: number): number {
    return num ** 3;
}) // 27
```

В TypeScript также можно делать **стрелочные функции**. Давайте посмотрим на примере. Пусть у нас есть следующая функция:

```
let func = function(num: number): number {
    return num ** 2;
}
```


Перепишем эту функцию на стрелочный вариант:

```
let func = (num: number): number => num ** 2;
```