**Введение в TypeScript**

Возможно, вы уже слышали про такой язык, как TypeScript. Разберёмся, почему он популярен среди фронтенд-сообщества.

Интернет кишит информацией, что с TypeScript вы будете писать меньше unit-тестов, а количество багов (особенно тех, что связаны с приведением типов) в приложениях уменьшится. Отчасти это так. Но главное преимущество TypeScript — удобство при разработке. Typescript присылает подсказки прямо во время написания кода. К примеру: «эта переменная — строка, не надо присваивать ей число» или «вы пытаетесь вызвать метод, которого не существует в этом классе, не надо так».

От JavaScript такой помощи в процессе написания кода не дождёшься: нужно собрать приложение и запустить его, чтобы увидеть, как оно падает с ошибкой.

В этом уроке посмотрим, как ведут себя TypeScript и JavaScript в одной и той же ситуации. Попытаемся передать в функцию переменную с неверным типом и сравним результат.

**JavaScript vs TypeScript**

JavaScript — язык с динамической типизацией. Это значит, что при инициализации переменным не задаётся какой-то конкретный тип и можно спокойно делать так:

Скопировать кодJSX

// js

let str = "это строка!";

str = 2; // а теперь число

В JavaScript str может превратиться в любой тип (не только в число) и никаких ошибок до поры до времени это не вызовет. Зато в момент выполнения программы вас будут ждать сюрпризы. JavaScript — интерпретируемый язык, так что все ошибки проявятся в самом конце.

Для наглядности создадим функцию и выведем текст в верхнем регистре:

Скопировать кодJSX

*// js*

function print(text) {

console.log(text.toUpperCase());

}

А затем вызовем её:

Скопировать кодJSX

print(str);

Вот только выше мы присвоили str = 2, и это уже не строка, а число. В этом случае нам не избежать ошибки, которая возникнет в момент выполнения программы: Uncaught TypeError: text.toUpperCase is not a function.

Теперь сравним, как в такой же ситуации поведёт себя TypeScript.

TypeScript в отличие от JavaScript — компилируемый язык со статической типизацией. Это значит, что он выявляет все подобные ошибки ещё на этапе написания кода.

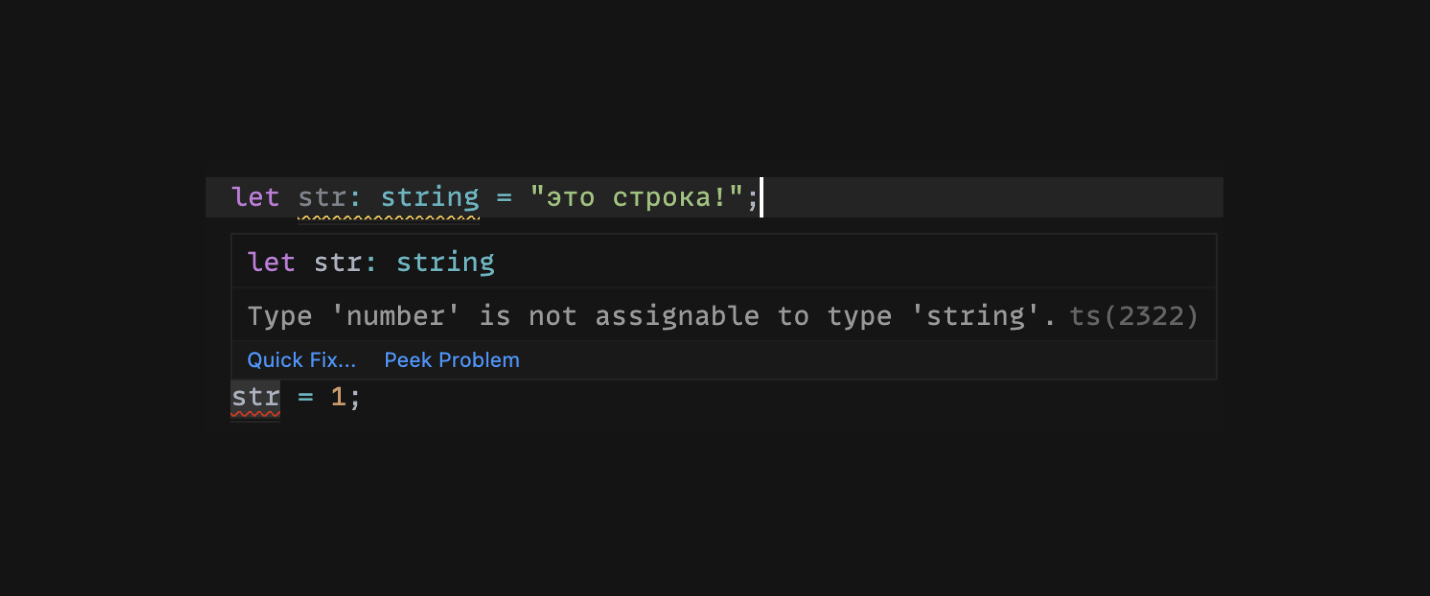
Начнём с объявления str. После названия переменной следует поставить : и указать её тип. В нашем случает это string:

Скопировать кодTSX

// ts

let str: string = "это строка!";

Теперь присвоить str число не получится. В IDE появится такая ошибка:



*Типизация в действии*

Соответственно, ошибки при передаче str в print не возникнет. TypeScript спасёт нас от провала ещё до сборки приложения!

Статическая типизация позволяет избежать проблем, которые в JavaScript связаны с приведением типов. Порой поведение языка с динамической типизацией не только ломает мозг разработчикам, но и может привести к ошибкам в процессе вычисления:

Скопировать кодJSX

// js

"0" == 0 // true

0 == [] // true

"0" == [] // Что же тут может быть? Конечно же false

// В TypeScript при попытках таких сравнений вы увидите ошибки в IDE

Как вы поняли, основная сила TypeScript — в статической типизации и подсказках при написании кода. Посмотрим, как выглядят эти подсказки в IDE и что ещё нового привнёс TypeScript.

**Моментальные подсказки**

Большинство существующих IDE поддерживают синтаксис языка, так что TypeScript оперативно сообщает об ошибках или выдаёт подсказку в процессе компиляции.

Например, если попытаетесь присвоить логической переменной строку:

Подсказывает в моменте.

Или если вы начнёте обращаться к полю, которого нет в объекте, увидите такое сообщение:



*Следит за переменными*

Кстати, как только вы напишете точку (Currency.), в подсказке отобразятся все доступные поля:

Можно не запоминать все поля в объекте.

При попытке переопределить тип полей объекта мы столкнёмся с той же ошибкой, что и при присваивании строке булевой переменной. IDE бдит за корректными типами и немножечко поругается:

Number, ты не пройдёшь!

Конечно, TypeScript не единственный в своём роде. Примерно вместе с ним начали появляться и другие альтернативы, например Dart или Flow. Но TypeScript оказался самым популярным из аналогов. TS лучше поддерживается разными библиотеками и популярными фреймворками: React, Angular, Vue. К тому же, вокруг TypeScript сложилось большое и прочное комьюнити.

В следующем уроке разберём основы TypeScript и немного попрактикуемся.

# Основы Typescript

Вы уже знаете, что TypeScript компилируется в JavaScript. В этом уроке разберём, как это происходит. Кроме этого, рассмотрим базовые типы TypeScript.

## Компиляция

Для начала установим TypeScript:

Скопировать кодBASH

npm install -g typescript

Теперь с помощью tsc (TypeScript Compiler) можно создать проект:

Скопировать кодBASH

tsc --init

В той папке, где мы выполним эту команду, появится файл tsconfig.json. \*\*Если откроем его, увидим compilerOptions с правилами компиляции проекта.

В compilerOptions есть два важных правила: target и strict. Разберём их в этом уроке. С остальными правилами вы можете ознакомиться [в официальной документации](https://www.typescriptlang.org/docs/handbook/compiler-options.html).

В target указывается версия ECMAScript. По умолчанию создаётся ES5. Если вы оставите это правило без изменений, то не сможете использовать фичи, которые появились в более поздних версиях, например в ES2020. Если же вы хотите попробовать все последние нововведения, выберите ESNext — она всегда указывает на следующую новую версию JavaScript.

А правило strict говорит о строгой проверке всех типов в приложении, без чего TypeScript не был бы собой.

Рассмотрим, как TypeScript превращается в JavaScript. Магия tsconfig распространяется на файлы с расширением .ts. Создадим рядом с tsconfig.json файл index.ts и для примера скопируем в него код функции, которая выполняет деление чисел:

Скопировать кодTYPESCRIPT

*// index.ts*

const divide = (dividend: number, divisor: number): number => {

return dividend / divisor;

};

Функция divide принимает два числа и возвращает их частное. Поэтому два её аргумента и результат вычисления имеют тип number. Тип переменной задаётся с помощью :. Получается, что в функцию divide можно передать только числа и в ответе получить число. Если передать в функцию другой тип, на этапе компиляции возникнет ошибка.

После того как мы добавили код функции в index.ts, скомпилируем TS в JS. Для этого нужно выполнить в терминале такую команду:

Скопировать кодBASH

tsc index.ts

Вот и всё, в папке появится index.js, который может быть выполнен в любом браузере:

Скопировать кодTYPESCRIPT

*// index.js*

var divide = function (dividend, divisor) {

return dividend / divisor;

};

С компиляцией разобрались, теперь перейдём к самой интересной части TypeScript — самим Types.

## Типы в TypeScript

Вы уже знаете, что в JavaScript типы делятся на примитивы (string, number, boolean, symbol, null, undefined и bigint) и объекты. Все эти типы существуют и в TypeScript — рассмотрим их в первую очередь. Затем разберём несколько новых типов, которые привнёс TS.

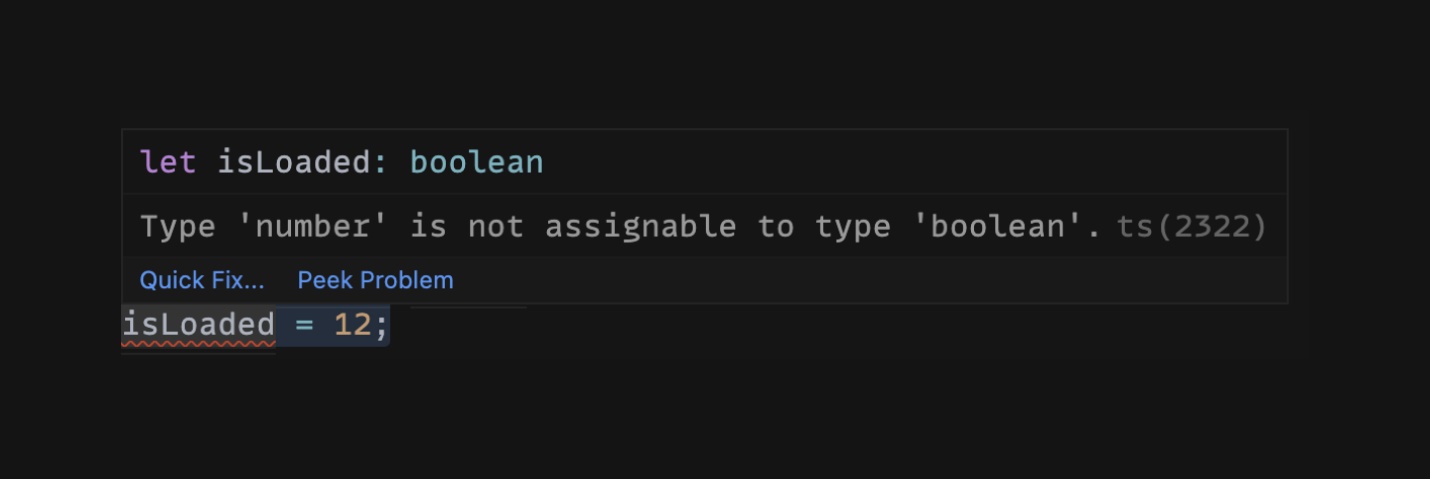
### Boolean

boolean один из простых типов, который задаётся так:

Скопировать кодTYPESCRIPT

let isLoaded: boolean = true;

Если попробуем присвоить isLoaded какой-то другой тип, например число, получим ошибку:



### Number

К типу number относятся и целые числа, и с плавающей точкой. Также вы можете задать число в любой системе счисления:

Скопировать кодTYPESCRIPT

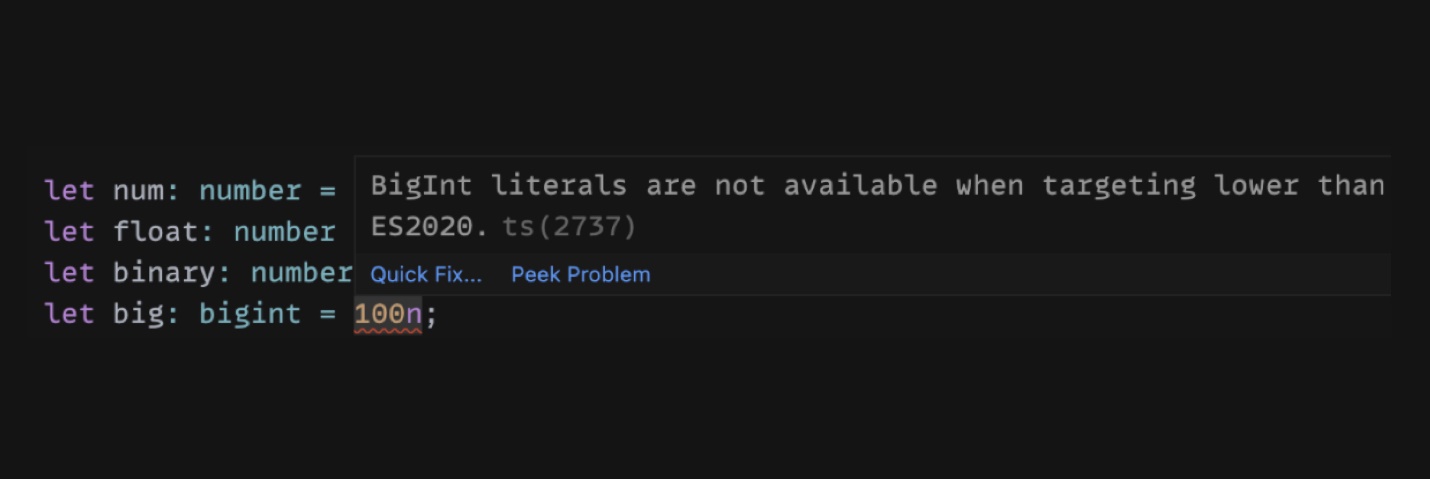
let num: number = 10;

let float: number = 10.5;

let binary: number = 0b101;

### BigInt

Ещё один числовой тип. Если вам необходимо его использовать, а в tsconfig.json указано "target": "es5", вы увидите ошибку:



Чтобы её устранить, нужно поменять запись на "target": "ES2020" или "target": "ESNext".

### String

Значение типа string можно указать в одинарных кавычках ', двойных " или косых `:

Скопировать кодTYPESCRIPT

const sentence: string = "Здесь могла быть ваша реклама";

const multiline: string = `Но тут только многострочный

текст и форматирование 1+2=${3}`;

### Symbol

Ещё один примитивный тип — symbol:

Скопировать кодTYPESCRIPT

const key: symbol = Symbol("Золотой ключик");

### Null и undefined

Примитивные значения, которые применяются для обозначения отсутствующего или неинициализированного значения:

Скопировать кодTYPESCRIPT

let undVar: undefined = undefined;

let nullVar: null = null;

Если у вас в конфиге не выставлен strictNullChecks или просто strict, эти типы можно присвоить переменным с любым другим типом. То есть не будет ошибкой сделать так:

Скопировать кодTYPESCRIPT

let str: string = "dndnnd";

str = undefined;

Но лучше не отключать strict — это позволит избежать лишних ошибок.

### Object

Объекты не относятся к примитивам. Чтобы их типизировать, можно использовать тип object:

Скопировать кодTYPESCRIPT

const book: object = {

pages: 684,

};

Но не всё так просто. Если попробовать определить количество страниц из book, получим ошибку на этапе компиляции:

Скопировать кодTYPESCRIPT

console.log(book.pages) *// Property 'pages' does not exist on type 'object'.*

В этом случае лучше указать тип не всего объекта, а отдельно всех его полей:

Скопировать кодTYPESCRIPT

const book: { pages: number } = {

pages: 684,

};

Его ещё называют анонимным типом. Внутри него в фигурных скобках указывают типизацию полей объекта. Так поле pages должно быть типа number. Теперь при обращении к pages ошибка не возникнет. А ещё не удастся присвоить полю другой тип.

### Array

Для объявления массива нужно указать тип его элементов и добавить []:

Скопировать кодTYPESCRIPT

const arrNum: number[] = [1, 2, 3]; *// массив чисел*

const arrStr: string[] = ["I", "am", "string"]; *// массив строк*

Так же можно объявить и многомерный массив:

Скопировать кодTYPESCRIPT

const arr: number[][] = [[1, 11, 101], [2, 22], [3]];

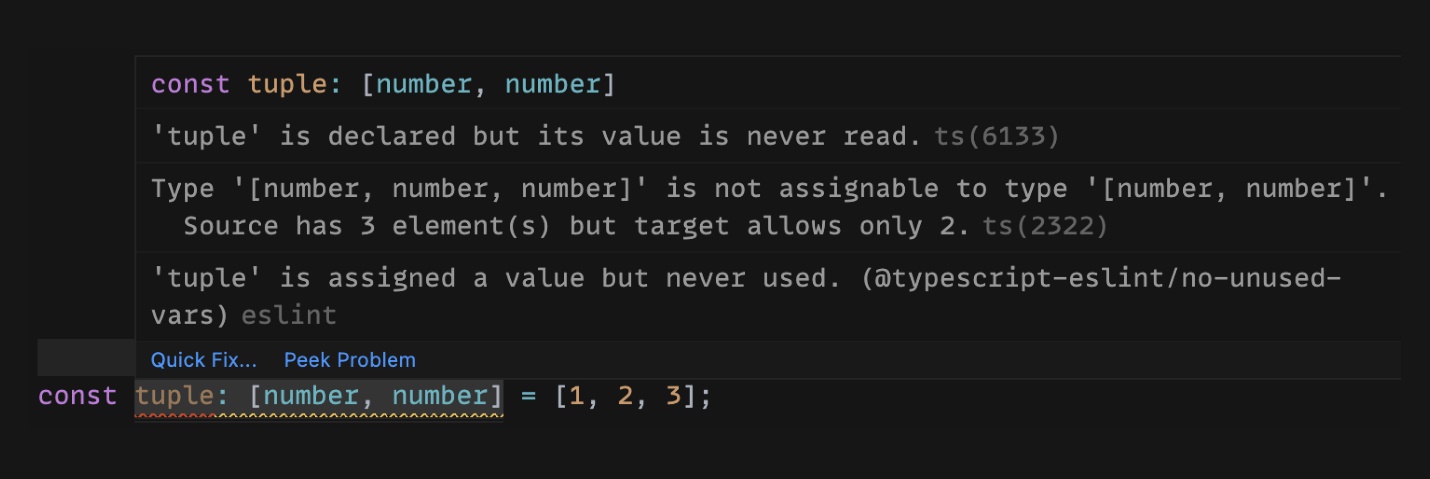
### Tuple

Кортеж, или tuple, — массив определённой длины.

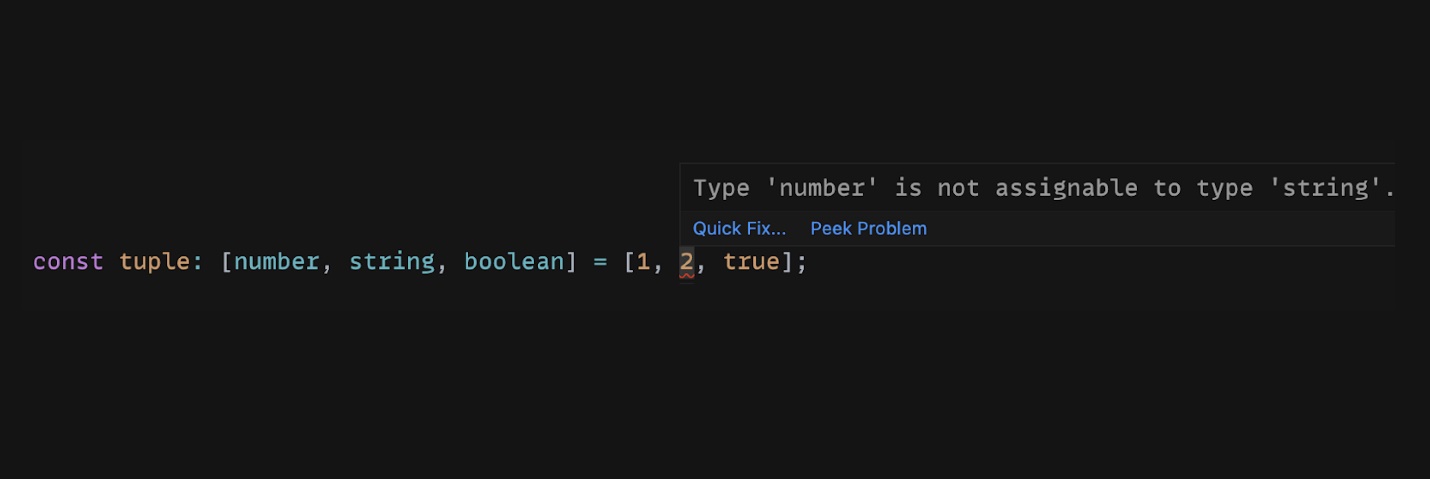
Скопировать кодTYPESCRIPT

const tuple: [number, number] = [1, 2];

tuple — кортеж из двух чисел, поэтому если задать больше элементов, чем указано в типе, отобразится ошибка:



Кортежи могут содержать разные типы. Для примера создадим кортеж из числа, строки и логического типа:



В этом случае возникнет ошибка. При объявлении типа мы указали, что под индексом 1 ожидается тип string, а сами записали число.

### Any

Тип, который указывает, что переменная может быть абсолютно любым типом. Иначе говоря, с типом any мы откатываемся назад к динамической типизации JavaSсript:

Скопировать кодTYPESCRIPT

let num: any = 12;

num = 'а вот уже я строка';

num = [];

num = 'и все же строка!';

Тип any можно использовать лишь в случае, когда нужно мигрировать большой проект с JavaScript на TypeScript. Сразу бывает сложно указать все необходимые типы, особенно в больших проектах, например интернет-магазинах. Но после того, как вы всё верно типизируете, не забудьте полностью очистить код от типа any.

### Unknown

В версии TypeScript 3.0 появился новый тип, который считается безопасным аналогом any — unknown. Чтобы в этом убедиться, сравним эти типы. Их поведение похоже — типу unknown может быть присвоен любой другой тип:

Скопировать кодTYPESCRIPT

let notSure: unknown = 0;

notSure = "окей, все-таки строка";

Но в отличие от any сам он не может быть присвоен:

Скопировать кодTYPESCRIPT

const str: string = notSure; *// Type 'unknown' is not assignable to type 'string'*

Ещё одно отличие — невозможность обратиться к полям или методам переменных, если их тип неизвестен. Создадим две функции и попробуем обратиться к абстрактному методу raveOn:

Скопировать кодTYPESCRIPT

const doSmth = (arg: any) => {

arg.raveOn();

};

const doSmthSafe = (arg: unknown) => {

arg.raveOn();

}

В doSmthSafe тип переменной arg неизвестен для TypeScript, поэтому он не рискнёт скомпилировать такой код. Ведь мы обращаемся к методу, которого может и не быть для типа данных, передаваемого при вызове doSmthSafe. Поэтому на этапе компиляции увидим ошибку:

Скопировать кодTYPESCRIPT

const doSmthSafe = (arg: unknown) => {

arg.raveOn(); *// (parameter) arg: unknown Object is of type 'unknown'.ts(2571)*

}

А вот с doSmth всё будет в порядке, пока мы не вызовем эту функцию. Например, с числом doSmth(999);. В этом случае ошибка отобразится только в момент выполнения, ведь у чисел нет функции raveOn.

Тип unknown ещё не раз нам пригодится, поэтому подробнее рассмотрим его в следующих уроках.

В этом уроке мы рассмотрели некоторые базовые типы TypeScript. Чтобы закрепить эти знания на практике, переходите к задачам.

Ваша задача:

* задать тип для HOME\_CRUMB;
* типизировать функцию isContainRoute, которая проверяет, содержится ли переданный route среди хлебных крошек;
* типизировать функцию removeRemainingCrumbs, которая удаляет часть хлебных крошек и возвращает новую цепочку навигации.

\* state -- history state of app

\* route -- url of a new breadcrumb \*/

export const isContainRoute = (state: { path: string, url: string, title: string }[], route: string): boolean =>

state.some(({ url }) => url === route);

/\*

\* state -- history state of app

\* url -- url of current breadcrumb \*/

export const removeRemainingCrumbs = (state: { path: string, url: string, title: string }[], url: string): { path: string, url: string, title: string }[] => {

const index = state.findIndex(({ url: route }) => route === url);

return state.slice(0, index);

};

export const HOME\_CRUMB: { path: string, url: string, title: string } = { path: '/', url: '/', title: 'Home' };

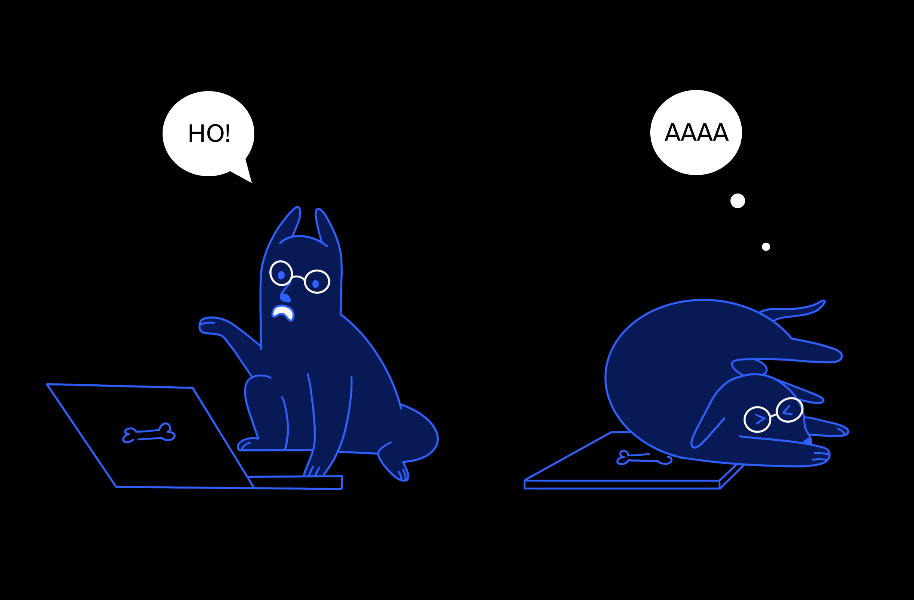
console.log(isContainRoute([HOME\_CRUMB], window.location.pathname));

# Больше типов богу типов

Вы уже знакомы с основными типами языка TypeScript, но это лишь верхушка айсберга. В этом уроке нырнём чуть глубже и познакомимся ещё с некоторыми основными типами, и даже сами создадим новый тип...

### Void

В TypeScript есть один интересный тип, который указывает на отсутствие любого типа.



Вот такой вот интересный тип, собачка!

То есть этот тип используют, чтобы показать, что функция ничего не возвращает:

Скопировать кодTYPESCRIPT

const printGreeting = (name: string): void => {

console.log(`Hi, ${name}`);

};

printGreeting("John Smith");

### Function type

В примере выше printGreeting — это переменная типа функции, вот только мы явно не указали этот тип, а отдельно типизировали аргументы анонимной функции (name: string) и возвращаемое значение void. Получилось, что printGreeting имеет следующий тип:

Скопировать кодTYPESCRIPT

(name: string) => void

Мы можем явно задать его вот так:

Скопировать кодTYPESCRIPT

const printGreeting: (name: string) => void = (name) => {

console.log(`Hi, ${name}`);

};

Такой вариант записи типа функции может быть полезен, например, когда вы задаёте тип для объекта, полем которого будет функция:

Скопировать кодTYPESCRIPT

const obj: { func: () => {} } = {

func: () => { console.log(‘DAS IST LOG’) },

};

### Enum

В TypeScript существует ещё один полезный тип — перечисление enum. Удивительно, что в JavaScript его до сих пор не завезли. Перечисление позволяет разработчику определить набор именованных констант.

Скопировать кодTYPESCRIPT

*// ts*

enum Currency {

RUB = "\u20BD",

DO = "\u0024"

}

То есть переопределить Currency.RUB = "Руб" нельзя.

Перечисления бывают числовыми, а также строковыми, как в примере выше. Числовое перечисление можно объявить так:

Скопировать кодTYPESCRIPT

enum Gender {

Male = 0,

Female = 1

}

А можно упростить запись и ничего не присваивать значениям Male и Female, за нас это по умолчанию сделает TypeScript.

Скопировать кодTYPESCRIPT

enum Gender {

Male, *// 0*

Female *// 1*

}

Для имплементации перечисления в JS нам бы пришлось создать объект и запретить изменять его поля:

Скопировать кодTYPESCRIPT

*// js*

const CurrencyJS = Object.freeze({

RUB: "\u20BD",

DOL: "\u0024"

});

### Literal Types

Представьте, что вы хотите затипизировать CSS-свойство position. Можно сделать так:

Скопировать кодTYPESCRIPT

let position: string = 'static';

Но это не совсем корректно, ведь при таком объявлении мы можем легко выставить position значение “где-то сбоку припёку”. Вот только такого CSS-значения не существует.

Тут на помощь приходят литеральные типы. С их помощью можно задать конкретную строку как тип. Теперь у position может быть только то значение, которое указано в типе:

Скопировать кодTYPESCRIPT

let position: 'static' = 'static';

position = 'absolute'; \**//  Type '"absolute"' is not assignable to type '"static"'.\**

Чтобы указать несколько значений типа, надо использовать объединение.

### Union types

TypeScript предоставляет для разработчика отличную возможность создавать новые типы путём комбинирования существующих, например, с помощью объединения.

Тип объединения образуется из двух или более типов, которые записываются через |.

Теперь мы можем указать несколько значений типа:

Скопировать кодTYPESCRIPT

let position: 'static' | 'absolute' | 'fixed' = 'static';

position = 'absolute'; \**//  теперь ошибки не будет!\* position принимает одно из 3 значений*

Поздравляем, вы только что создали новый тип — 'static' | 'absolute' | 'fixed'!

Посмотрим, как ещё можно использовать объединения. Создадим функцию, которая может работать со строками и числами:

Скопировать кодTYPESCRIPT

const printId = (id: number | string) => {

console.log(`ID = ${id}`);

};

В printId мы можем передать строку или число:

Скопировать кодTYPESCRIPT

printId("bond-007");

printId(777);

Если будем передавать значения других типов, например null, получим ошибку: Argument of type 'null' is not assignable to parameter of type 'string | number'.

Допустим, мы хотим изменить функцию и выводить id в верхнем регистре:

Скопировать кодTYPESCRIPT

const printId = (id: number | string) => {

console.log(`ID = ${id.toUpperCase()}`);

};

Вот только метод toUpperCase() доступен строкам и отсутствует у чисел. При компиляции мы получим ошибку. Чтобы её избежать, нужно проверить, что id будет именно строкой. Сделать это можно с помощью оператора typeof, который вам уже знаком из JavaScript:

Скопировать кодTYPESCRIPT

const printId = (id: number | string) => {

if (typeof id === "string") {

console.log(`ID = ${id.toUpperCase()}`);

} else {

console.log(id);

}

};

В этом уроке вы познакомились с новыми типами и даже создали свой собственный! Немного задач и можно идти дальше

**ООП в TypeScript**

Пока мы говорили только про подсказки и статическую типизацию, но ещё TypeScript предоставляет расширенное (по сравнению с JavaScript) объектно-ориентированное программирование (ООП).

В языке появляются модификаторы доступа public, private, protected. С их помощью можно инкапсулировать поля и методы классов. Кроме этого, образуются интерфейсы — абстракции для описания полей классов. О них мы поговорим в следующих уроках. А пока рассмотрим модификаторы на примере класса проигрывателя:

Скопировать кодTYPESCRIPT

class Player {

model: string = "";

private settings: string = "";

constructor() {

*//*

}

protected soundCheck() {

*// ...*

}

public play() {

*// ...*

}

pause() {

*// ...*

}

}

Создадим экземпляр класса Player:

Скопировать кодTYPESCRIPT

const player = Player();

И посмотрим, к каким методам дают или ограничивают доступ модификаторы:

* Модификатор public. По умолчанию все члены класса — публичные. Можно вызвать player.play() и даже player.pause(), хоть в pause модификатор public и не указан явно. Ещё в классе Player публичными являются constructor и поле model. В JavaScript до ES2020 все члены классов — публичные.
* Модификатор private. А вот к settings можно обратиться только внутри класса Player. В JavaScript только начиная с ES2020 появился приватный модификатор — #.
* Модификатор protected тоже ограничивает доступ, но не так радикально, как private. Если мы захотим унаследовать новый класс от класса Player — виниловый проигрыватель, то в этом классе будет доступен soundCheck. А вот вызвать метод через экземпляр класса player.soundCheck() уже не получится. Другими словами, protected — модификатор, который открывает доступ для наследников.

Теперь вы знаете какой модификатор поможет скрыть методы от лишних глаз, пора познакомиться с интерфейсами!

# Использование interface и type в Typescript

Когда вы листаете понравившуюся книгу в магазине, обращаете внимание на её обложку, содержание и, возможно, аннотацию. Они помогают понять, о чём книга. У объектов в TypeScript тоже может быть такое содержание. Для описания структуры объекта можно использовать interface или type.

## Интерфейс

Interface — абстракция, которая описывает структуру объекта. Разберёмся на примере. Создадим объект, скажем, книгу hitchhikerGalaxy:

Скопировать кодTYPESCRIPT

const hitchhikerGalaxy = {

author: "Д.Адамс",

title: "Автостопом по галактике",

pages: 684,

read: (): string => {

return "Я читаю, пусть весь мир подождёт";

}

}

Теперь типизируем этот объект. Это можно сделать двумя способами: написать hitchhikerGalaxy: object или добавить анонимный объект. Первый вариант не очень удобен, потому что типы полей объекта будут нетипизированы. Так что воспользуемся вторым способом:

Скопировать кодTYPESCRIPT

const hitchhikerGalaxy: {

author: string;

title: string;

pages: number;

read: () => string;

} = {

Если нам потребуется ещё один объект с точно таким же типом, придётся писать идентичный анонимный тип. Чтобы избежать дублирования кода, создадим интерфейс, в котором опишем типы полей книги:

Скопировать кодTYPESCRIPT

interface IBook {

author: string;

title: string;

pages: number;

read: () => string;

}

Обычно названия интерфейсов указывают с I. Это негласное правило, придерживаться которого необязательно. Но мы рекомендуем ему следовать, ведь тогда из названия сразу понятно, что перед нами интерфейс. А это помогает не запутаться в случае, когда мы объявляем и применяем тип в разных файлах.

Укажем IBook при инициализации переменной:

Скопировать кодTYPESCRIPT

const hitchhikerGalaxy: IBook = {

author: "Д.Адамс",

title: "Автостопом по галактике",

pages: 684,

read: (): string => {

return "Я читаю, и пусть весь мир подождёт";

}

}

Всё, что мы описали в интерфейсе, обязательно должно содержаться в объекте. Теперь у всех полей строго определены типы. И полю с типом string мы можем присвоить только строковое значение. Вот только этим присваиванием можно переопределить текущее значение поля, например, поменять название книги на другое:

Скопировать кодTYPESCRIPT

hitchhikerGalaxy.title = "Преступление и наказание";

Но это уже вполне тянет на нарушение авторских прав. От наказания нас может спасти ключевое слово readonly. Изменим наш интерфейс и запретим менять авторов и название книги:

Скопировать кодTYPESCRIPT

interface IBook {

readonly author: string;

readonly title: string;

pages: number;

read: () => string;

}

Теперь при попытке изменить название получим ошибку: Cannot assign to 'title' because it is a read-only property.

Интерфейс IBook описывает поля, которые могут содержаться в hitchhikerGalaxy. Ни больше, ни меньше. Те поля, что помечены readonly, не переопределяются. Так что при попытке добавить новую пару ключ-значение получим ошибку:

Скопировать кодTYPESCRIPT

hitchhikerGalaxy.answer = 42; *// ошибка --> Property 'answer' does not exist on type 'IBook'*

IBook содержит общие характеристики книги: имя автора, название произведения и количество страниц. Разберёмся, что делать, если потребуется добавить что-то специфичное для конкретной книги, к примеру ответ на главный вопрос жизни, который дал искусственный интеллект — answer. Для этого в интерфейс IBook можно добавить поля и пометить их как необязательные:

Скопировать кодTYPESCRIPT

interface IBook {

readonly author: string;

readonly title: string;

pages: number;

read: () => string;

answer?: number;

}

Необязательное поле следует помечать знаком ?.

По задумке поле answer должно относиться только к книге «Автостопом по галактике». Но на практике получается не так: если мы добавим ещё и книгу о Золушке, могут возникнуть проблемы с архитектурой — опциональное поле будет относиться и к новой книге тоже.

Добавим answer другим способом. Интерфейсы могут расширять друг друга. Так что создадим новый интерфейс для нашей книги и расширим его с помощью IBook:

Скопировать кодTYPESCRIPT

interface IHitchhikerGalaxy extends IBook {

answer: number;

}

IHitchhikerGalaxy содержит объявление answer и всех полей, которые есть в IBook. При инициализации hitchhikerGalaxy укажем IHitchhikerGalaxy и добавим поле answer:

Скопировать кодTYPESCRIPT

const hitchhikerGalaxy: IHitchhikerGalaxy = {

...

answer: 42,

}

Теперь поле answer содержится только в объектах типа IHitchhikerGalaxy, а в объектах типа IBook его не будет.

Мы рассмотрели вариант с типизацией объекта с помощью интерфейса. Но есть и другой вариант: присвоить анонимному типу псевдоним, чтобы повторно использовать в приложении и импортировать в разные файлы.

## Псевдоним типа

Добавим анонимному типу название с помощью ключевого слова type:

Скопировать кодTYPESCRIPT

type TBook = {

readonly author: string;

readonly title: string;

pages: number;

read: () => string;

};

Если интерфейсы обычно начинаются с буквы I, то типы — с буквы T.

Псевдоним можно задать абсолютно любому типу. К примеру, удобно делать это для типа функций. Задать псевдоним для функции read можно так:

Скопировать кодTYPESCRIPT

type TStringFunc = () => string;

type TBook = {

...

read: TStringFunc;

};

Теперь тип TStringFunc можно использовать для типизации любой функции без аргументов, которая возвращает строку.

Посмотрите внимательно на новый тип TBook и интерфейс IBook. Объявление интерфейса и псевдонима очень схоже за исключением пары синтаксических особенностей:

Скопировать кодTYPESCRIPT

*// ключевое слово interface*

interface IBook { *// нет знака =*

readonly author: string;

readonly title: string;

pages: number;

read: TStringFunc;

answer?: number;

} *// нет ;*

*// ключевое слово type*

type TBook = { *// есть знак =*

readonly author: string;

readonly title: string;

pages: number;

read: TStringFunc;

}; *// есть ;*

Но на синтаксисе отличия не заканчиваются. Есть и технические особенности, которые подскажут, когда лучше использовать type, а когда interface. Разберём это подробнее в следующем уроке.

**Дженерики в TypeScript**

Представьте, что вы решили посадить растение. Для этого вам потребуются горшок с землёй и семена. В один и тот же горшок можно посадить практически любое растение: фикус, кактус или помидоры черри. Так же и с дженериками, или обобщёнными типами, в TypeScript. Компонент — один, а типы данных, с которыми он работает, — разные. Дженерики можно применять с функциями, интерфейсами и классами. Рассмотрим, как это сделать.

**Generic Functions**

Напишем функцию, которая принимает разные типы аргументов:

Скопировать кодTYPESCRIPT

const growPlant = <T>(plant: T): T => {

return plant;

}

В треугольных скобках указывается обобщённый тип, которому можно придумать любое название. Для упрощения мы использовали букву T.

При объявлении функции это название не имеет значения, но оно должно указывать, что на входе и выходе функции ожидается одинаковый тип (string, number или любой другой). Этот тип мы укажем, когда будем вызывать функцию.

Создадим пару растений и посмотрим, как вызвать функцию growPlant с созданными типами:

Скопировать кодTYPESCRIPT

type Tomato = 'Tomato';

type Daisy = 'Daisy';

const tomato: Tomato = 'Tomato';

const daisy: Daisy = 'Daisy';

С каждым из этих растений мы можем вызвать функцию growPlant. К примеру, посадим помидор:

Скопировать кодTYPESCRIPT

growPlant<Tomato>(tomato);

Без дженерика эту же запись можно представить так:

Скопировать кодTYPESCRIPT

const growPlant = (plant: Tomato): Tomato => {}

Но если бы мы решили посадить ещё десяток растений, нам пришлось бы создать столько же функций под каждый тип:

Скопировать кодTYPESCRIPT

const growPlant = (plant: Tomato): Tomato => {}

const growPlant2 = (plant: Daisy): Daisy => {}

const growPlant3 = (plant: string): string => {}

...

бесконечность — не предел

...

С обобщённым типом вы экономите время, ведь теперь нужно написать значительно меньше кода, даже если вы решили обзавестись целой плантацией.

Пока growPlant принимает всего один аргумент, и его тип соответствует возвращаемому типу функции. Вот только аргументов функции может быть сколько угодно, притом все с разными типами. Усложним функцию:

Скопировать кодTYPESCRIPT

const growPlant = <T, U, R>(plant: T, sunshine: U): R => {}

Здесь мы указали типы двух аргументов T, U и тип, который вернёт функция, — R.

Посмотрим, как вызвать growPlant с тремя разными типами. Передадим в «горшок» growPlant «семечко» с типом Seed. При правильном «поливе» Water оно превратится в «растение» Plant:

Скопировать кодTYPESCRIPT

growPlant<Seed, Water, Plant>(...)

Ну, или мы засушим его на «солнце» UV и получим гербарий:

Скопировать кодTYPESCRIPT

growPlant<Seed, UV, Herbarium>(...)

**Обобщённые типы в интерфейсах**

Тот же «цветочный горшок» мы можем описать интерфейсом:

Скопировать кодTYPESCRIPT

interface FlowerPot<TPlant> {

plant: TPlant;

growPlant: () => TPlant;

}

Пока в FlowerPot мы всегда должны передавать какой-то тип:

Скопировать кодTYPESCRIPT

const fw: FlowerPot<string> = ...

Если его не передать, тип переменной будет any, а не FlowerPot:

Скопировать кодTYPESCRIPT

const fw: FlowerPot = ...

*// при наведении на fw увидим const fw: any*

Можно защититься от any — при объявлении дженерика указать тип по умолчанию:

Скопировать кодTYPESCRIPT

interface FlowerPot<TPlant = {}> {

plant: TPlant;

growPlant: () => TPlant;

}

Теперь при наведении на fw мы увидим, что переменная действительно типа FlowerPot:

Скопировать кодTYPESCRIPT

const fw: FlowerPot = ...

*// const fw: FlowerPot<{}>*

**Обобщённые типы в классах**

Чтобы рассмотреть обобщения в контексте классов, сделаем класс «подоконник» Windowsill, на который поставим «горшок с цветком» FlowerPot:

Скопировать кодTYPESCRIPT

class Windowsill<TMaterial, TPlant> implements FlowerPot<TPlant> {

dimension: Array<number>; *// размерности подоконника [1.8, 0.6, 0.09]*

plant: TPlant;

private material: TMaterial; *// материал подоконника*

constructor(plant: TPlant) {

this.plant = plant;

}

growPlant(): TPlant {

return water(this.plant); *// просто добавь воды, и цветок вырастет*

}

}

В обобщении мы указали два типа:

* TMaterial — тип для поля material;
* TPlant — тип для интерфейса.

Теперь мы можем создать много разных подоконников. Например, дома будет гранитный, а цветок на нём может быть любой, главное, чтобы он был строкой. А на работе сделаем пластиковый подоконник с таким же цветком:

Скопировать кодTYPESCRIPT

const homeWindowsill = new Windowsill<'Granite', string>('Rose');

const jobWindowsill = new Windowsill<'Plastic', string>('Rose');

В этом уроке вы познакомились с обобщёнными типами и узнали, как их можно применять с функциями, интерфейсами и классами. Главная сила дженериков в том, что они помогают избегать дублирования кода.

Пока мы самостоятельно создавали дженерики, но TypeScript настолько хорош, что имеет в своём арсенале ещё и встроенные дженерики. Они сильно облегчат нам жизнь, познакомимся с ними в следующем уроке.

# Встроенные дженерики

В предыдущем уроке мы реализовали кастомные дженерики. Но TypeScript предоставляет широкий перечень полезных обобщённых типов. Рассмотрим некоторые из них в этом уроке.

### Array<T>

Обратите внимание на поле dimension в классе Windowsill:

Скопировать кодTYPESCRIPT

class Windowsill<TMaterial, TPlant> implements FlowerPot<TPlant> {

dimension: Array<number>;

...

dimension — массив чисел. В предыдущих уроках мы объявляли его так: dimension: number[]. В этом — воспользовались обобщением.

Array<T> — внутренний тип, предоставляемый TypeScript. Array<number> идентичен записи number[]. В обоих случаях мы получаем массив чисел, отличие лишь в синтаксисе объявления типа.

### ReadonlyArray<T>

Если вдруг мы не хотим, чтобы размер подоконника можно было менять, то есть сделать массив dimension доступным только для чтения, следует воспользоваться дженериком ReadonlyArray<T>:

Скопировать кодTYPESCRIPT

dimension: ReadonlyArray<number>; *// аналогично readonly number[];*

### Promise<T>

Ещё один полезный встроенный дженерик — Promise<T>. Он используется для указания возвращаемого типа в функциях async/await. Например, для типизации функции запроса к API:

Скопировать кодTYPESCRIPT

const loadData = async (): Promise<TResponse> => await fetch('https://backend', ...)

Или если просто требуется создать промис:

Скопировать кодTYPESCRIPT

const promise = new Promise<string>((resolve) => {

resolve("Всё просто, указана строка, резолвим строку")

});

При создании мы сразу указали тип, который ожидаем получить в процессе исполнения промиса.

### Omit<T>

Ещё один популярный встроенный дженерик — Omit. С его помощью можно опустить поля типа или интерфейса.

Допустим, у нас есть тип TBook, и мы хотим сделать из него тип TAuthor. Тогда при объявлении нового типа следует убрать поля title и pages. Для этого воспользуемся Omit:

Скопировать кодTYPESCRIPT

type TBook = {

readonly authorName: string;

readonly authorLastName: string;

readonly title: string;

pages: number;

};

*// вторым аргументом Omit указываются названия полей, которые надо опустить*

type TAuthor = Omit<TBook, 'title' | 'pages'>

Новый тип TAuthor теперь выглядит так:

Скопировать кодTYPESCRIPT

type TAuthor = {

readonly authorName: string;

readonly authorLastName: string;

};

### Pick<T>

Дженерик-антагонист Omit. С его помощью можно выбрать интересующие нас поля интерфейса.

Тот же тип TAuthor можно получить с помощью Pick:

Скопировать кодTYPESCRIPT

*// вторым аргументом Pick указываются названия полей, которые надо получить*

type TAuthor = Pick<TBook, 'authorName' | 'authorLastName'>

Полезно использовать, когда из интерфейса со множеством полей надо выбрать пару значений для создания нового типа.

### ReturnType<T>

ReturnType<T> создаёт тип, который возвращает переданная в него функция. Например, ReturnType пригодится для типизации thunk, но об этом мы поговорим, когда разберём Redux. Напишем функцию getId, которая вернёт объект id:

Скопировать кодTYPESCRIPT

function getId() => {

return { id: "bond-007" };

}

Теперь тип функции getId надо передать в ReturnType. Вот только мы не знаем её тип. Получить его можно с помощью typeof:

Скопировать кодTYPESCRIPT

*// при наведении на t TS покажет --> type t = { id: string; }*

type t = ReturnType<typeof getId>

TypeScript содержит гораздо больше встроенных дженериков, чем мы изучили в этом уроке. Самостоятельно почитать про некоторые из них вы можете [в официальной документации языка](https://www.typescriptlang.org/docs/handbook/utility-types.html).

export type TUser = {

readonly id: number;

readonly password: string;

readonly email: string;

readonly name: string;

};

export type TRawUser = Omit<TUser, 'id'> & { \_id: number };

Promise<>

//types.ts

export type TUser = {

readonly id: number;

readonly password: string;

readonly email: string;

readonly name: string;

};

export type TRawUser = Omit<TUser, 'id'> & { \_id: number };

export type TAffiliations = {

readonly name: string;

readonly city: string;

readonly country: string;

};

export type TPrize = {

readonly year: string;

readonly category: string;

readonly share: string;

readonly motivation: string;

readonly affiliations: ReadonlyArray<TAffiliations>;

};

export type TCountry = {

readonly name: string;

readonly code: string;

};

export type TLaureate = {

readonly id: string;

readonly firstname: string;

readonly surname: string;

readonly born: string;

readonly died: string;

readonly bornCountry: string;

readonly bornCountryCode: string;

readonly bornCity: string;

readonly diedCountry: string;

readonly diedCountryCode: string;

readonly diedCity: string;

readonly gender: string;

readonly prizes: ReadonlyArray<TPrize>

};

export type TResponseBody<TDataType> = {

success: boolean;

user?: TDataType;

message?: string;

headers?: Headers;

};

export interface CustomResponse<T> extends Body {

readonly headers: Headers;

readonly ok: boolean;

readonly redirected: boolean;

readonly status: number;

readonly statusText: string;

readonly trailer: Promise<Headers>;

readonly type: ResponseType;

readonly url: string;

clone(): Response;

json(): Promise<T>;

}

//Index.ts

import { CustomResponse, TResponseBody, TRawUser } from "./types";

export const loginRequest = async (form: {

email: string;

password: string;

}): Promise<CustomResponse<TResponseBody<TRawUser>>> =>

await fetch('https://cosmic.nomoreparties.space/login', {

method: 'POST',

mode: 'cors',

cache: 'no-cache',

credentials: 'same-origin',

headers: {

'Content-Type': 'application/json'

},

redirect: 'follow',

referrerPolicy: 'no-referrer',

body: JSON.stringify(form)

});

loginRequest({ email: 'spacedancer1337@nomoreparties.space', password: 'ToPsEcReT007421337'}).then(res => {

return res.json();

}).then(data => {

console.log(data);

})

# Отличия interface от type

В этом уроке разберём четыре особенности, которые отличают псевдоним типа type от интерфейса interface.

Сделаем это на примерах из предыдущего урока. Чтобы было удобно сравнивать, снова объявим тип TBook:

Скопировать кодTYPESCRIPT

type TBook = {

readonly author: string;

readonly title: string;

pages: number;

read: () => string;

};

А также интерфейс IBook:

Скопировать кодTYPESCRIPT

interface IBook {

readonly author: string;

readonly title: string;

pages: number;

read: () => string;

}

## Пересечение типов (Intersection Types)

Мы говорили, что интерфейсы могут расширять друг друга. Благодаря этой особенности мы могли бы создать интерфейс, который содержит все поля IBook и какие-то новые:

Скопировать кодTYPESCRIPT

interface IHitchhikerGalaxy extends IBook {

answer: number;

}

Типы в отличие от интерфейсов не расширяются, и записать type THitchhikerGalaxy extends TBook не удастся. Но в случае, если мы работаем с псевдонимами типов, а в TBook нет поля answer, мы можем создать новые типы путём пересечения существующих:

Скопировать кодTYPESCRIPT

type THitchhikerGalaxy = TBook & {

answer: number;

};

С помощью & создаётся пересечение типов. С интерфейсами оно тоже работает. Можно создать интерфейс и не расширять его, а объединить с другим:

Скопировать кодTYPESCRIPT

interface IHitchhikerGalaxy {

answer: number;

};

const hitchhikerGalaxy: IHitchhikerGalaxy & IBook = {

На деле мы получили новый тип из двух интерфейсов. Присвоим ему псевдоним:

Скопировать кодTYPESCRIPT

type THitchhikerGalaxyBook = IHitchhikerGalaxy & IBook;

Получается, что при пересечении интерфейсов мы получаем три типа: THitchhikerGalaxyBook, IHitchhikerGalaxy и IBook. А при расширении одного интерфейса другим — два типа: IHitchhikerGalaxy и IBook, но IHitchhikerGalaxy будет содержать в себе все поля IBook.

## Добавление полей в существующий тип

В псевдоним типа нельзя добавлять новое поле, тогда как в интерфейсе это обычное дело. Таким способом мы можем добавить answer к объявленному ранее интерфейсу IBook. Допустим, IBook объявлен в файле book.ts, а answer мы добавим в component.ts. При этом добавлении IBook изменится глобально.

Скопировать кодTYPESCRIPT

*// component.ts*

interface IBook {

answer: number;

};

*// теперь IBook содержит author, title, pages, read и answer*

С псевдонимами типов так сделать не получится. Если попробуем повторно объявить псевдоним, получим ошибку:

Скопировать кодTYPESCRIPT

type TBook = { *// Identifier 'TBook' has already been declared*

answer: number;

};

## Реализация интерфейса классом

В предыдущем уроке мы рассматривали интерфейс для типизации объектов, но у него есть ещё одно применение, которое отличает его от type. В ООП классы могут реализовывать интерфейсы. То есть вы описываете в интерфейсах поля и методы, которые должен содержать класс (как будто описываете содержание книги). Реализовать же псевдоним типа классом — нельзя.

Создадим класс Book и реализуем интерфейс IBook. Класс должен содержать все поля и методы, описанные в интерфейсе. Добавим конструктор класса, который на вход принимает title, author и pages:

Скопировать кодTYPESCRIPT

interface IBook {

readonly author: string;

readonly title: string;

pages: number;

read: () => string;

}

class Book implements IBook {

public readonly author: string;

public readonly title: string;

public pages: number;

constructor(title: string, author: string, pages: number) {

this.title = title;

this.author = author;

this.pages = pages;

}

public read() {

return "Я читаю, и пусть весь мир подождёт";

}

}

*// создадим экземпляр класса Book*

const book = new Book("Автостопом по галактике", "Д.Адамс", 684);

Обратите внимание на модификаторы членов класса — они все публичные. Определить поля, описанные в интерфейсах как protected или private, нельзя.

Что и когда использовать в разработке — решать вам. Если нужно описать структуру класса, тут точно не обойтись без interface. Если требуется описать поля объекта и тип не надо расширять, достаточно применить type.

Теперь вы точно знаете ответ на главный вопрос жизни, Вселенной, интерфейсов, псевдонимов типов и всего такого. Закрепим знания на практике.

# Связанные типы (mapped types)

Вы уже познакомились с основами TypeScript: типизацией, интерфейсами и дженериками. Пришло время погрузиться в этот язык программирования глубже. В этом уроке узнаем, как объявить тип для объекта, если названия его полей неизвестны.

Иногда при типизации объектов нам неизвестны названия ключей, к примеру ответ от API. Тогда мы можем указать для ключа тип. Напишем в квадратных скобках условное название name и тип ключа:

Скопировать кодTYPESCRIPT

interface IAnswer {

[name: string]: number;

}

Теперь создадим объект с типом IAnswer. В качестве ключа можно указать любое строчное значение:

Скопировать кодTYPESCRIPT

const ans: IAnswer = {

"key": 0,

"other\_key": 1,

"error\_key": "eeeeeerror" *// Type 'string' is not assignable to type 'number'.*

}

Но значение должно быть числом, иначе получаем ошибку.

Пока любая строка может являться ключом IAnswer. Но можно и ограничить названия ключей через знакомый нам [тип union (объединение)](https://practicum.yandex.ru/learn/react/courses/69d8dabd-d91d-441b-9b22-4638e613e705/sprints/9588/topics/26bbc34b-989a-4301-b4b9-d69392d19414/lessons/22aef4fd-2d32-4972-b132-2171029f16f7/). Для примера создадим тип, ключами которого будут значения литерального типа TFontWeight:

Скопировать кодTYPESCRIPT

*// добавим псевдоним для строковых здачений font-weight*

type TFontWeight = 'bold' | 'bolder' | 'lighter' | 'normal';

Напишем тип TFontWeightRecord. В квадратных скобках при помощи оператора in укажем, что ключ должен быть типа TFontWeight:

Скопировать кодTYPESCRIPT

type TFontWeightRecord = { [fw in TFontWeight]: number };

Объявление [fw in TFontWeight] можно сделать только в type, что также отличает его от interface.

Теперь можно создать объект с типом TFontWeightRecord:

Скопировать кодTYPESCRIPT

const fw: TFontWeightRecord = {

bold: 700,

bolder: 900,

normal: 400,

lighter: 300

};

Мы рассмотрели итерацию по литеральному типу TFontWeight. В случае, если мы захотим в качестве ключей для создания нового типа использовать такой тип:

Скопировать кодTYPESCRIPT

type TFontWeightObj = {

bold: number;

bolder: number;

normal: number;

lighter: number;

};

При попытке объявить тип, как и TFontWeightRecord, получим ошибку:

Скопировать кодTYPESCRIPT

type TFontWeightObjRecord = { [fw in TFontWeightObj]: number };

*// Type 'TFontWeightObj' is not assignable to type 'string | number | symbol'. Type 'TFontWeightObj' is not assignable to type 'string'.*

Мы получили ошибку из-за того, что итерация может производиться только для [объединённых строковых литеральных типов](https://practicum.yandex.ru/trainer/react/lesson/7ac5f495-8fd1-41ad-9e3e-ce846f18e312/task/81265a45-3a6f-43e0-81bb-ed1b94189f33), а TFontWeightObj таким не является. Но есть решение — создать литеральный тип из названия ключей TFontWeightObj. Сделать это можно с помощью оператора keyof:

Скопировать кодTYPESCRIPT

type TKeys = keyof TFontWeightObj;

*// если наведём на TKeys, то увидим:*

*// Alias for: keyof TFontWeightObj*

*// Initial type: "bold" | "bolder" | "normal" | "lighter"*

Добавлять псевдоним TKeys необязательно, можно сразу написать:

Скопировать кодTYPESCRIPT

type TFontWeightObjRecord = { [fw in keyof TFontWeightObj]: number };

Теперь ошибки не будет, и мы можем объявить объект c типом TFontWeightObjRecord:

Скопировать кодTYPESCRIPT

const fwObj: TFontWeightObjRecord = {

bold: 100,

bolder: 100,

normal: 100,

lighter: 100

};

Название ключей мы взяли из TFontWeightObj. А в качестве значения задали тип number. Но мы могли бы указать и тип значения из TFontWeightObj. Чтобы его получить, нужно использовать обращение по индексу. Перепишем объявление TFontWeightObjRecord:

Скопировать кодTYPESCRIPT

type TFontWeightObjRecord = { [fw in keyof TFontWeightObj]: TFontWeightObj[fw] };

Теперь вы знаете, как создать литеральный тип на основе ключей другого типа. И что по ключу можно получать его значение — тип ключа.

Кроме этого, разобрались, как объявлять тип или интерфейс, когда неизвестны названия ключей. Чтобы закрепить этот материал, переходите к задачам.