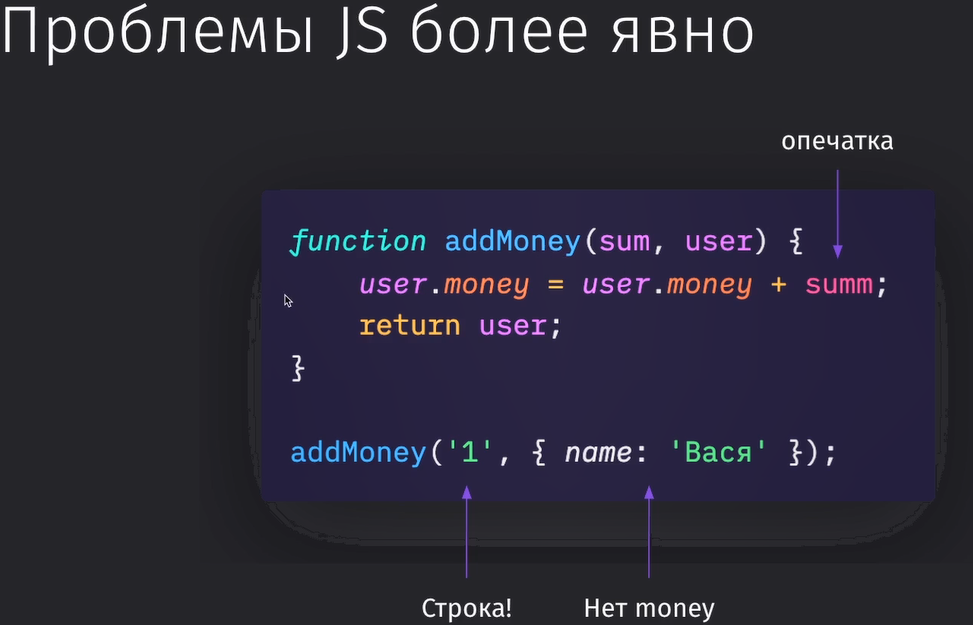
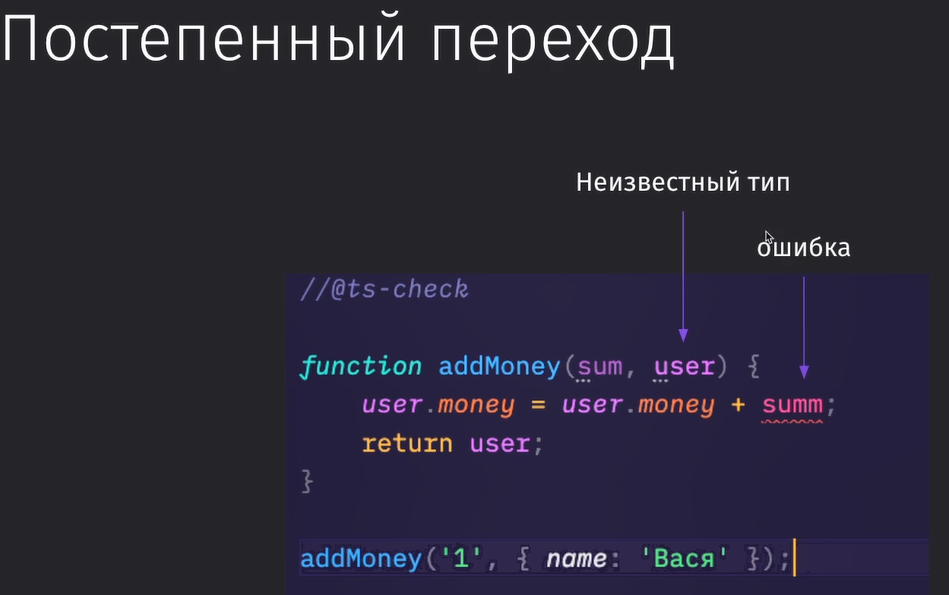


**Отсутствие Type Safety.**

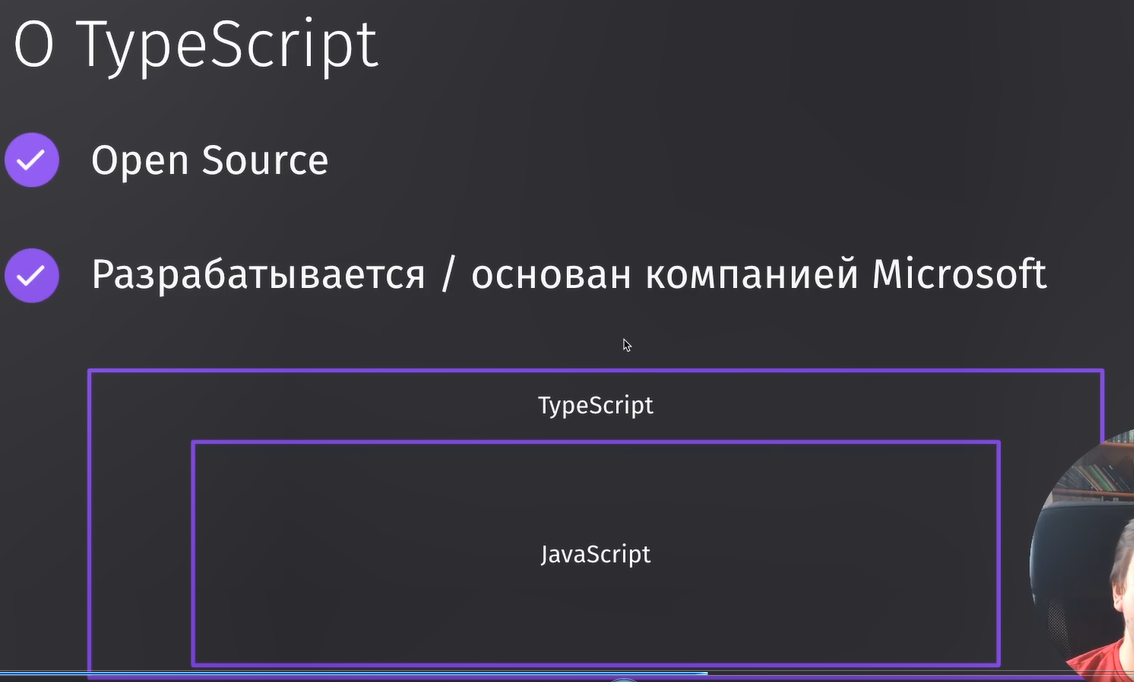
Не проверяется корректность приведения типов, не проверяем наличие переменных, которые у нас присутствуют, но можно решить с помощью ESLint.

**Динамически типы** (не строгая типизация) – мы можем приплюсовывать строку к числу, и т.д.

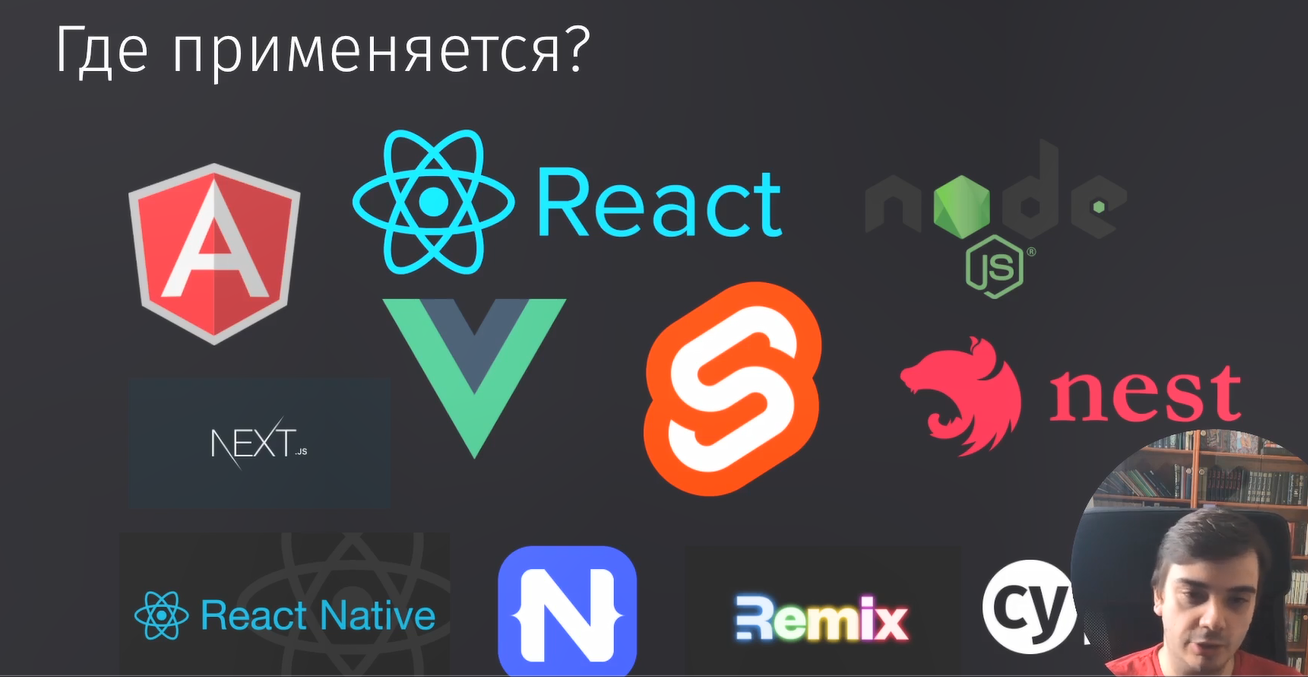


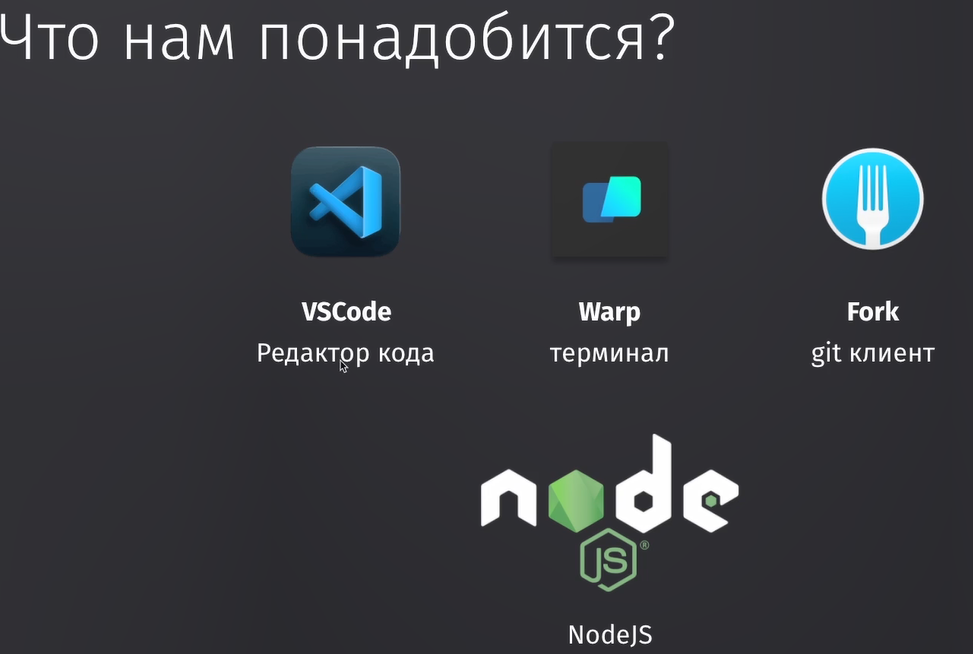






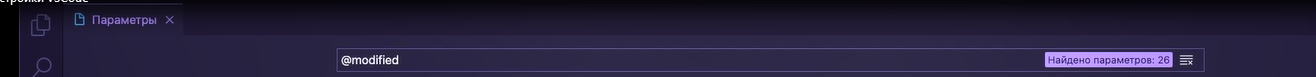
1-52



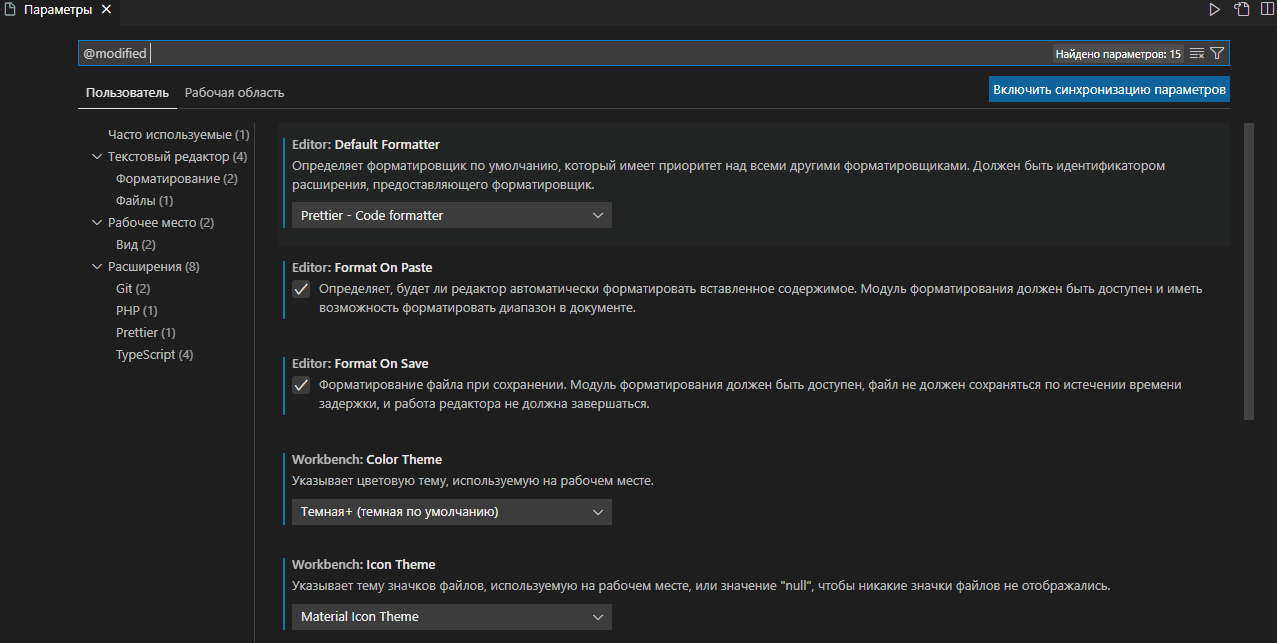


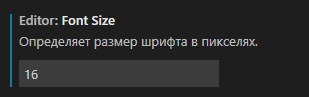
https://github.com/coreybutler/nvm-windows

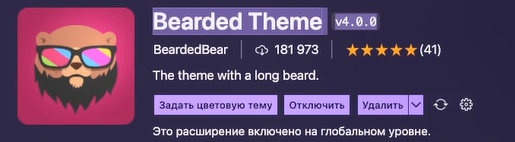




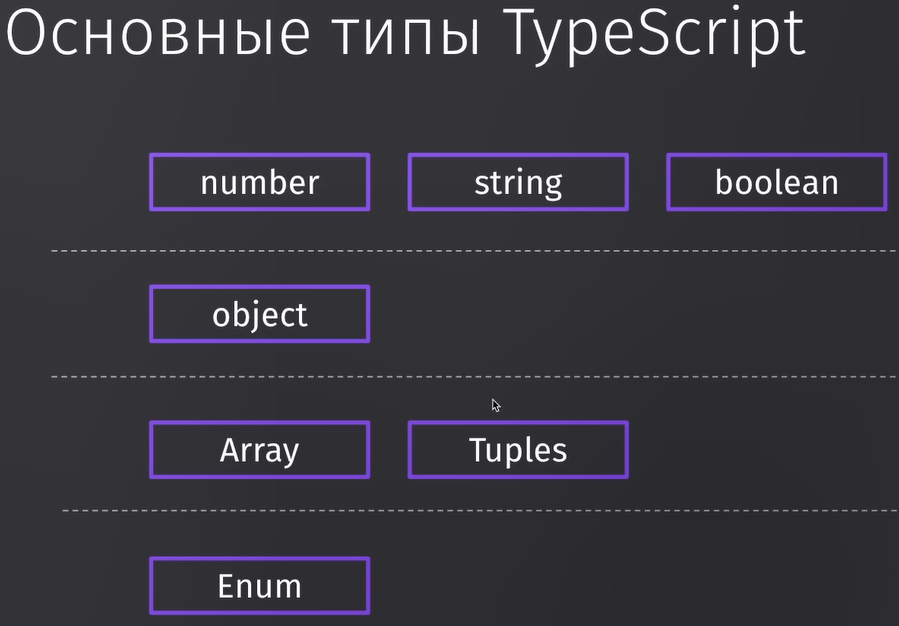
Просмотр измененных параметров:







**Основные типы**



Установим глобально typescript:

**npm i -g typescript**

**tsc –help**

**typescript compailer**

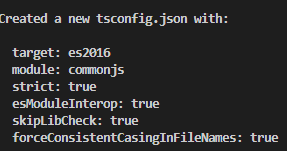
Это компилятор, который позволяет брать из typescript код, компиоировать его в js для дальнейшего исполнения.

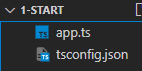
Имеет возможность создавать конфиги, создадим минимальный.



Создает предзаполненный ts конфиг.



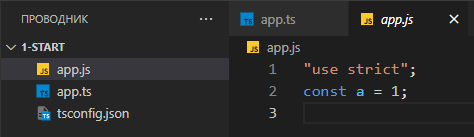




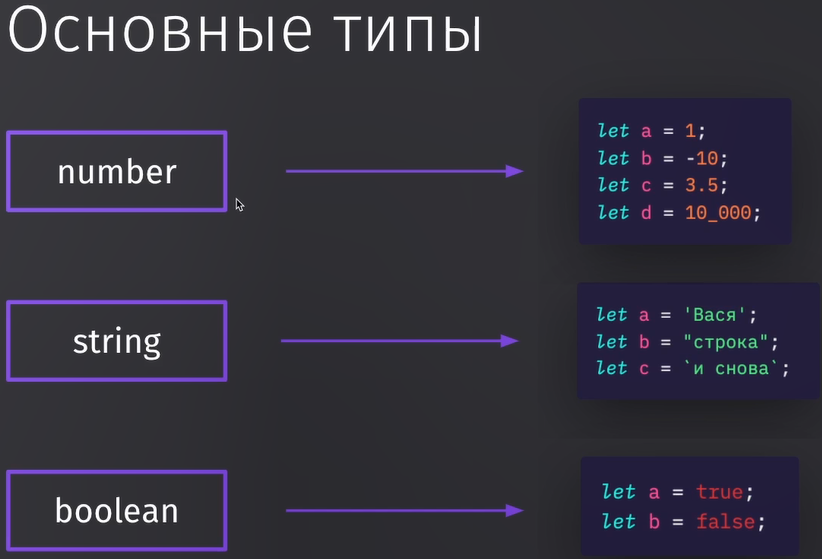
const a = 1;

Скомпилируем этот файл:





**Использование типов**



const a = 1;

let revenue = 1000;

let bonus = 500;

let res = revenue + bonus;

console.log(res);

Результат:

"use strict";

const a = 1;

let revenue = 1000;

let bonus = 500;

let res = revenue + bonus;

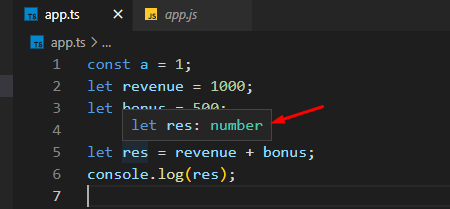
console.log(res);

Исполним:



TS может типо выводить, так мы можем их указывать явно.

При наведении мыши TS анализирует тип данных:



Сделаем ошибку:

let bonus = '500';

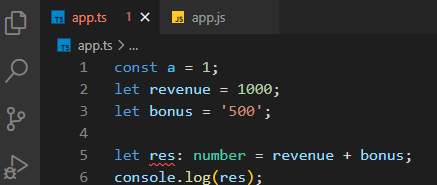


Наш результат должен быть числом.

Поэтому запишем тип через :

let res: number = revenue + bonus;

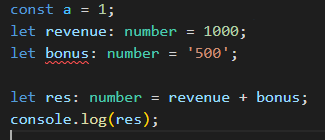
Код подсветился красным:



Можем и здесь задать число:

let revenue: number = 1000;

let bonus: number = '500';



Теперь у нас ошибка переместилась на уровень bonus.

let revenue: number = 1000;

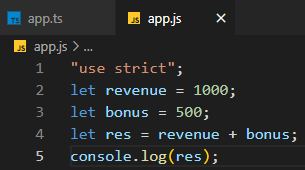
let bonus: number = 500;

let res: number = revenue + bonus;

console.log(res);

Теперь все будет работать корректно.

После компиляции информация о типах в js файле пропала. Но можем сохранить с помощью мапов.

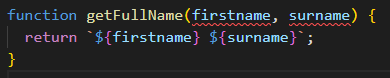


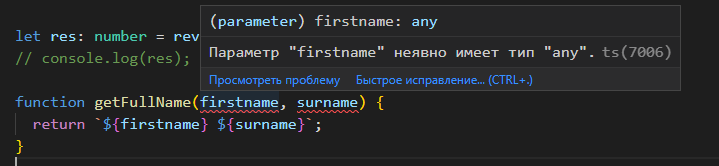
let c: string = 'sdf';

let d: boolean = true;

**Типы в функциях**

Сделаем функцию, которая будет выдавать имя + фамилию.





Дефолтная установка tsconfig

**any** – означает любой тип.

Это особый тип в TS, который обеспечивает совместимость в JS

Но при этом он задан не явно.

Мы можем так записать:

function getFullName(firstname: any, surname) {

  return `${firstname} ${surname}`;

}

**any** – валидный тип, обозначающий все, что угодно.

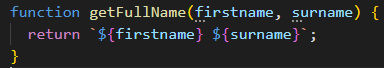
Или изменим tsconfig.



Тоесть если есть неявный any, то будет ошибка.

 "noImplicitAny": false

В итоге ошибка ушла:



Укажем явно возвращаемый формат функции:

function getFullName(firstname: string, surname: string): string {

  return `${firstname} ${surname}`;

}

Аналогично в виде стрелочной функции:

const getFullNameAll = (firstname: string, surname: string): string=> {

  return `${firstname} ${surname}`;

}

Проверка типов, которые находятся в runtime

function getFullNameAny(firstname, surname) {

  return `${firstname} ${surname}`;

}

console.log(getFullNameAny(true, false));



Сделаем првоерку на JS:

function getFullNameAny(firstname, surname) {

  if (typeof firstname !== 'string') {

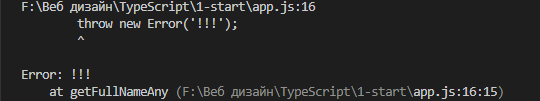
    throw new Error('!!!');

  }

  return `${firstname} ${surname}`;

}

console.log(getFullNameAny(true, false));



В чем проблема такого потхода в TS?

TS нам и нужен, чтобы обеспечить безопасность типов.

Такие проверки в TS бесполезны.

JS динамически типизированный язык

TS статически типизированный

В JS мы можем менять на лету типы, прибавляем число к строке, у нас автоматически происходит конвертация числа в строку.

В TS типы статические. И если мы зададим тип, то без явной конвертации этого типа мы не сможем работать с этим типом.

Поэтому явно пропишем типы, и опцию **"noImplicitAny": true**

Т.е. мы не сможем указать явные типы any.

function getFullNameAny(firstname: string, surname: string): string {

  //   if (typeof firstname !== 'string') {

  //     throw new Error('!!!');

  //   }

  return `${firstname} ${surname}`;

}

// console.log(getFullNameAny(true, false));

console.log(getFullNameAny('Виталий', 'Галата'));



**Объекты**

Напишем функцию, которая будет принимать пользователя.

// типизация объекта

function getInfoPeople(userEntity: {

  firstname: string;

  lastname: string;

}): string {

  return `${userEntity.firstname} ${userEntity.lastname}`;

}

const user = {

  firstname: 'Vitaliy',

  lastname: 'Galata',

  city: 'Nikolaev',

  age: 37,

};

console.log(getInfoPeople(user));

Т.е. мы можем передать помимо двух обязательных свойств и другие свойства.

Наш объект в итоге шире.

Аналогично работают объекты со вложенностью:

const user = {

  firstname: 'Vitaliy',

  lastname: 'Galata',

  city: 'Nikolaev',

  age: 37,

  skills: {

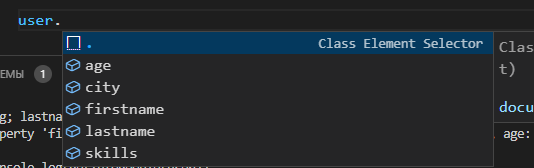
    dev: true,

    devops: true,

  },

};

Большое преимущесвто tsc над js, что мы не сможем опечаться, когда захотим обратится к свойству объекта.



Упражнение.

Нам приходит json:

**{**

**"officeId": 45,**

**"isOpened": false,**

**"contacts": {**

**"phone": "+79100000000",**

**"email": "my@email.ru",**

**"address": {**

**"city": "Москва"**

**}**

**}**

**}**

Нужно его типизировать.

let userInfo: {

  officeId: number;

  isOpened: boolean;

  contacts: {

    phone: string;

    email: string;

    address: {

      city: string;

    };

  };

};

Мы можем его сравнить с объектом:

let userInfo: {

  officeId: number;

  isOpened: boolean;

  contacts: {

    phone: string;

    email: string;

    address: {

      city: string;

    };

  };

} = {

  officeId: 45,

  isOpened: false,

  contacts: {

    phone: '+79100000000',

    email: 'my@email.ru',

    address: {

      city: 'Москва',

    },

  },

};

И оно будет валидно.

**Массивы**

// типизируем массив - указывем явно - массив строк

const skills: string[] = ['Dev', 'DevOps'];

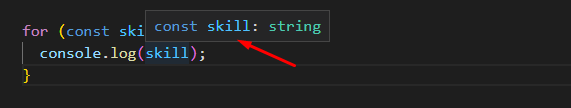
Но если нужен массив неоднородный:

// типизируем не однородный массив

const skills: any[] = ['Dev', 'DevOps', 1];

**any не стоит использовать.**

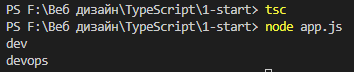
Каждый элемент массива будет строкой:



for (const skill of skills) {

  console.log(skill.toLowerCase());

}



Явно укажем входящий параметр для метода **filter():**

const filterArray = skills.filter((s: string) => s !== 'DevOps');

console.log(filterArray); // [ 'Dev', 'Testing'

const filterArrayMap = skills

  .filter((s: string) => s !== 'DevOps')

  .map((s) => s + '!');

console.log(filterArrayMap); // [ 'Dev!', 'Testing!' ]

В итоге во всех фильтрах мы правильно использовали поведение типов:

const filterArrayMapReduce = skills

  .filter((s: string) => s !== 'DevOps')

  .map((s) => s + '! ')

  .reduce((a, b) => a + b);

console.log(filterArrayMapReduce); // Dev! Testing!

**Tuples**

**Кортежы или tuples.**

const a: string[] = ['str1', 'str2', 1];

Есть проблема, пытаемся в масив добавить число.

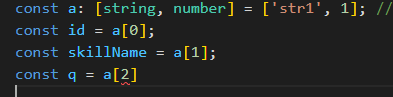


const a: [string, number] = ['str1', 1]; // typel это массив ограниченой длины, где каждый элемент типизированный

const id = a[0];

const skillName = a[1];

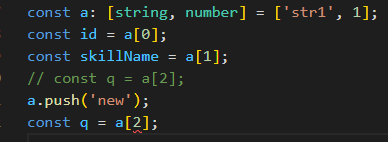
**При этом мы не можем обращаться к элементу в не описания:**



Но мы можем пушить:

a.push('new');

Но даже после этого мы не можем обратиться ко второму элементу массива.



Также в typles работают теже деструктуризации, что и в массивах.

const ski: [number, string] = [1, 'Dev'];

const [idSkill, nameSkill] = ski;

console.log(idSkill, nameSkill); // 1 Dev

Можем использовать spread оператор на типах.

// использование spread оператора на boolean типах

const arr: [number, string, ...boolean[]] = [1, 'sdf', true, true, false];

const arrNew: [number, string, ...boolean[]] = [1, 'sdf'];

**Readonly**

**Проблема модификаций кортежев и массивов.**

// переопределим наш нулевой элемент кортежа двойкой:

const newSkillsNew: [number, string] = [1, 'Dev'];

newSkillsNew[0] = 2;

console.log(newSkillsNew); // [ 2, 'Dev' ]

Воспользуемся модификатором в TS readonly:

const newSkillsNew: readonly [number, string] = [1, 'Dev'];



Данный кортеж не модифицированный.



// альтернатива : string[] - использование Generic

const skillsGeneric: Array<string> = ['Dev', 'DevOps']; // Array<string> - это всего лишь массив строк

console.log(skillsGeneric); // [ 'Dev', 'DevOps' ]

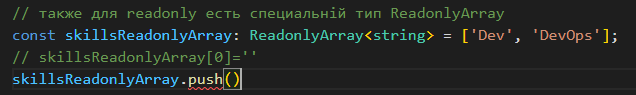
// также для readonly есть специальный тип ReadonlyArray

const skillsReadonlyArray: ReadonlyArray<string> = ['Dev', 'DevOps'];

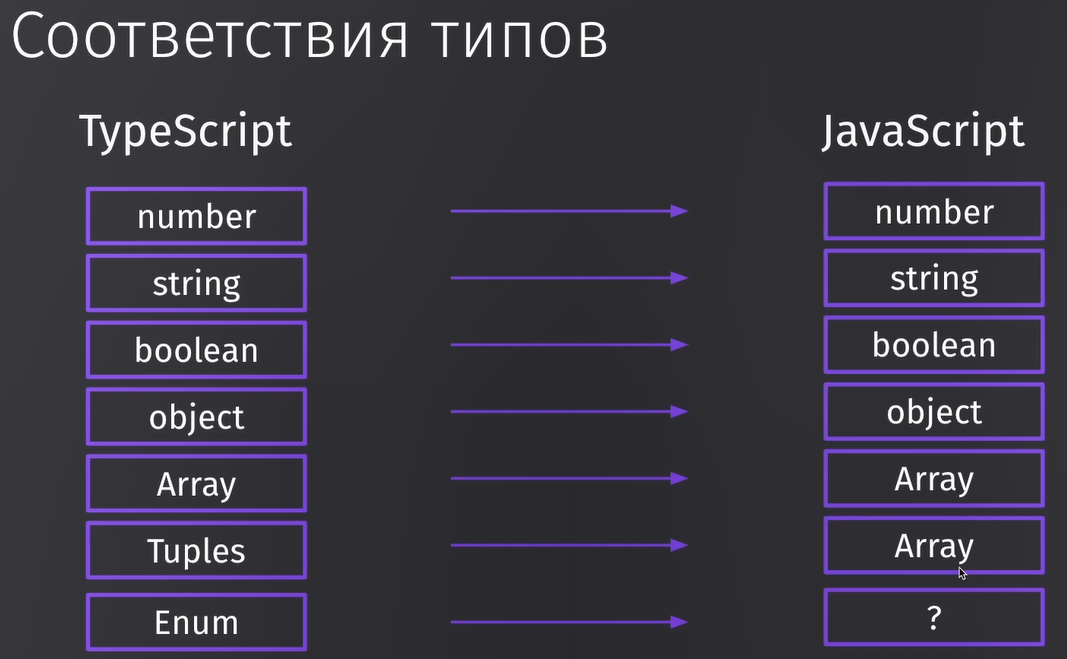


В JS мы не можем зафиксировать длину, модификацию массива.

Пушить тоже не можем:



**Enums**



const res = {

  message: 'Платеж успешный',

  statusCode: 1

}

// 1 - успех

// 2 - в процессе

// 3 - отклонен

**Если мы будем модифицировать объект, то этому статусу можем присвоить другое значение.**

Или мы можем:

const res = {

  message: 'Платеж успешный',

  statusCode: 'success'

}

// 1 - успех

// 2 - в процессе

// 3 - отклонен

if (res.statusCode === 'succes') {

  // спец. сделали опечатку

}

Такие проблемы и решаем Enum.

В обычном js мы сдеалил бы константу:

if (res.statusCode === SUCCESS) {

  // спец. сделали опечатку

}

Так бы мы избавились от опечаток.

Но нам нужно создавать наборы констант для каждого статуса.

enum StatusCode {

  SUCCESS,

  IN\_PROCESS,

  FAILED,

}

const res = {

  message: 'Платеж успешный',

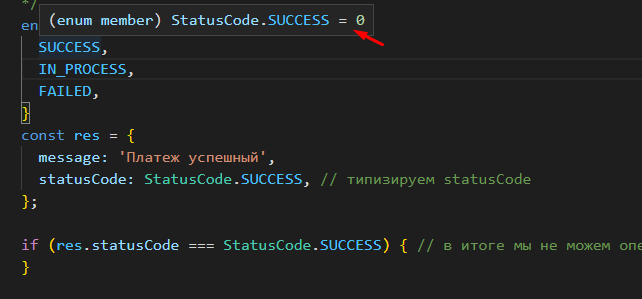
  statusCode: StatusCode.SUCCESS, // типизируем statusCode

};

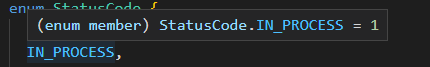
if (res.statusCode === StatusCode.SUCCESS) { // в итоге мы не можем опечататься, получить другие значения

}

Как понять – чему равен success ?



Следующий 1 и т.д.



Если присвоим 1, то увидим, что остальные увеличиваются:

enum StatusCode {

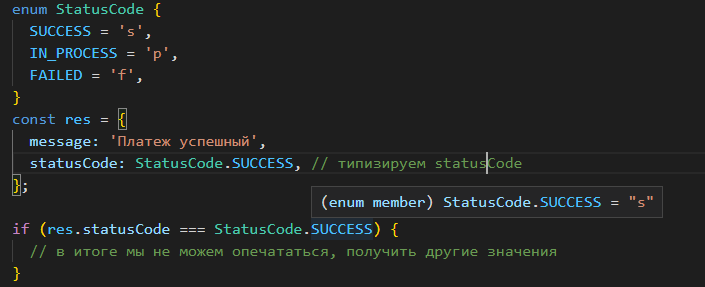
  SUCCESS = 1,

  IN\_PROCESS,

  FAILED,

}

Enum может быть и строковым.



**Гетерогенные enum**

Может быть или числом, или строкой.

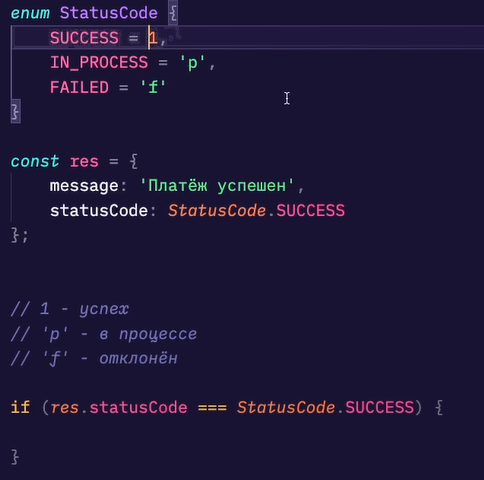
enum StatusCode {

  SUCCESS = 1,

  IN\_PROCESS = 'p',

  FAILED = 'f',

}



function action(status: StatusCode) {

  // хотим принимать енамы этого типа: StatusCode

  console.log('success');

}

action(StatusCode.SUCCESS); // access

action(1) // тоже будет валидное значение, так как SUCCESS = 1 присутствует в SUCCESS

// вот так будет проблема, там как енам гетерогенный, но продолжает быть числовым

action('p')

action(StatusCode.IN\_PROCESS); // лучше вот так

С помощью enum можно сделать справочник кодов ответов. Каких-то велечин.

Или роли для пользователей.

enum Roles {

  ADMIN,

  USER,

}

Enum могут быть рассчетными:

enum Roles {

  ADMIN = 1,

  USER = ADMIN \* 2,

}

Даже функцию можно передать:

function compute() {

  return 3;

}

enum Roles {

  ADMIN = 1,

  USER = compute(),

}

Енамы похожи на объекты.

enum Roles {

  ADMIN = 1,

  USER = compute(),

}

function test(x: { ADMIN: number }) {

}

test(Roles);

Тоесть мы в функцию test передали enum, и он ведет себя как объект.

enum Roles {

  ADMIN = 1,

  USER = compute(),

}

function test(x: { ADMIN: number }) {

  console.log('x', x); // x { '1': 'ADMIN', '3': 'USER', ADMIN: 1, USER: 3 }

}

test(Roles);

Константные енамы

const enum Roleses {

  ADMIN = 1,

  USER = 2,

}

Но после компиляции мы не увидем ничего.

Так как мы объявляем роли как константы, то компилятор ищет все места, где упоминается Roleses и заменяет его на константы.

Так как таких мес нет, то ничего и не будет.

const res2 = Roleses.ADMIN;

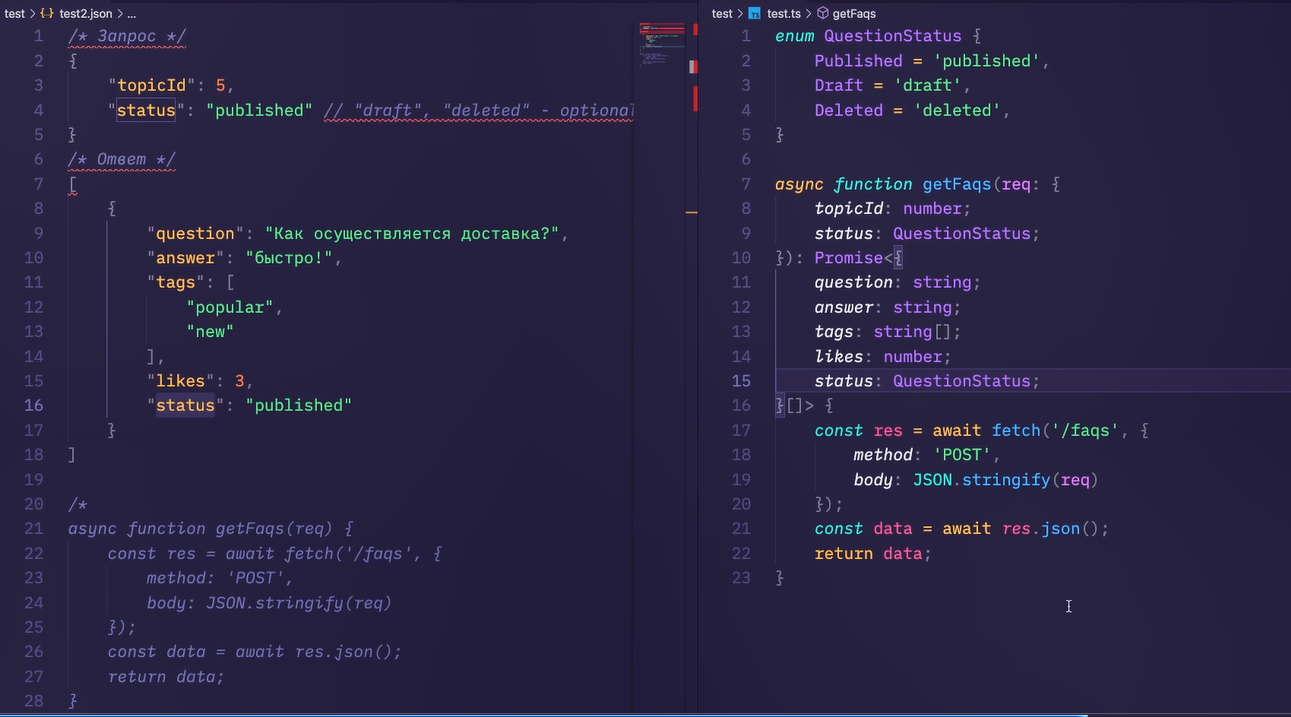
То после компиляции:

const res2 = 1 /\* Roleses.ADMIN \*/;

Енамы это функции в JS после компиляции если мы определили обычный енам. Константные енамы, то это просто константы, которые подставляются в нужные места, где у нас прилетели константы.

**Типизация функции**

У нас есть запрос и ответ.



enum QuestionStatus { // сделаем через enum, так как имеем ограниченное число ответов

  Published = 'published',

  Draft = 'draft',

  Deleted = 'deleted',

}

async function getFaqs(req: {

  topicId: number;

  status: QuestionStatus;

}): Promise<

  {

    // злест пишем, что функция возвращает - возвращает массив объектов - в конце пишем []

    question: string;

    answer: string;

    tags: string[]; // массив стрингов

    likes: number;

    status: QuestionStatus;

  }[]

> {

  const res = await fetch('/faqs', {

    method: 'POST',

    body: JSON.stringify(req),

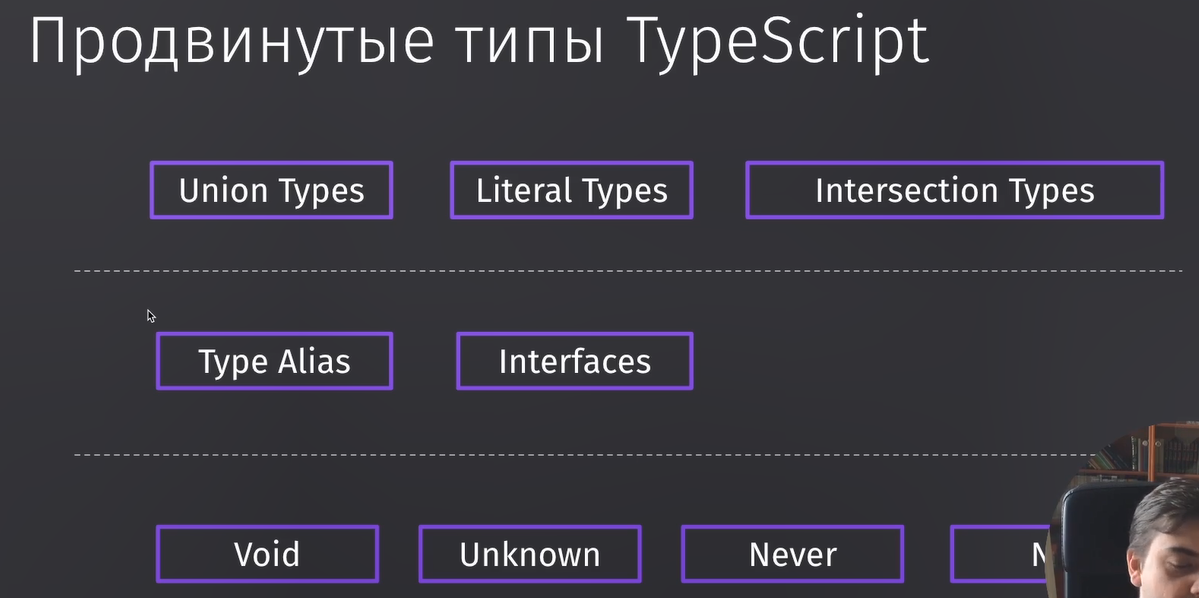
  });

  const data = await res.json();

  return data;

}

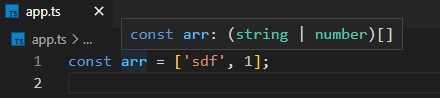
**Продвинутые типы**



**Union**

В переменной могут находится различные типы при различных обстоятельствах.

const arr = ['sdf', 1];



Выводится тип Union между string/number.

const arr = ['sdf', 1];

// функция, которая будет логировать переданный нами id

// принимает Union тип

function logId(id: string | number) {

  console.log(id);

}

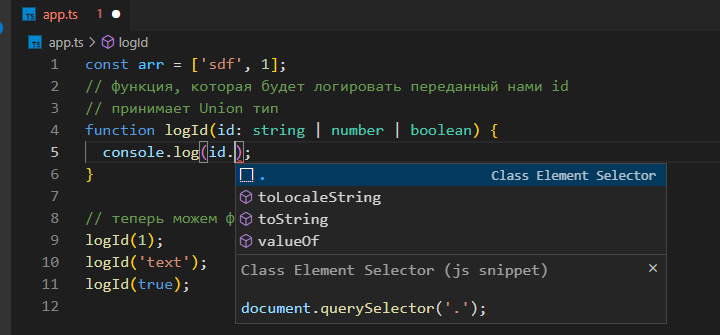
// теперь можем функцию вызвать либо с number либо с string

logId(1);

logId('text');

Но теперь нужно привести все в нижний регистр:

Но здесь ничего нет:



Это обозначает, что id может принимать 3 типа и он union тип.

С Union типом можно уметь работать раздельно.

Это называется сужжение (marring) типов.

Сужжение тимпов позволяет нам сделав runtime проверку, применить ограничение на наш тип уже в рамках ts.

function logId(id: string | number | boolean) {

  if (typeof id === 'string') {

    console.log(id);

  }else if (typeof id === 'number') {

     console.log(id);

  }else{

    console.log(id)

  }

}

// работа с cложными типами

// функция будет принимать либо одну ошибку либо массив ошибок

function logError(err: string | string[]) {

  if (Array.isArray(err)) {

    // если это массив

    console.log(err);

  } else {

    console.log(err);

  }

}

Это работает и с объкектами, с классами и т.д.

// принимаем объект, у которого есть либо свойство a/b

function logObject(obj: { a: number } | { b: number }) {

  // оператор in - проверяет, а есль ли тот или иной ключ в объекте

  if ('a' in obj) {

    console.log(obj.a);

  } else {

    console.log(obj.b);

  }

}

// функция, которая принимает multiply dist

// в обоих случаях есть тип string

function logMultipleIds(a: string | number, b: string | boolean) {

  // чтобы сделать сужжение в таких случаях

  if (a === b) {

    // а по типу равно b ?

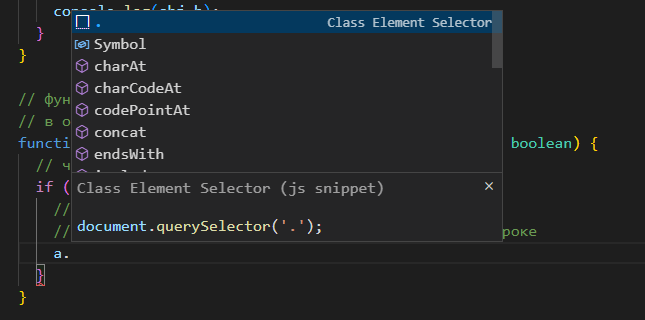
    // то мы можем обращаться к a/b обращаться как к строке

    a.

  }

}

В итоге нам доступны все методы строки:



// функция, которая принимает multiply dist

// в обоих случаях есть тип string

function logMultipleIds(a: string | number, b: string | boolean) {

  // чтобы сделать сужжение в таких случаях

  if (a === b) {

    // а по типу равно b ?

    // то мы можем обращаться к a/b обращаться как к строке

    return a.length;

  } else {

    console.log(a);

  }

}

console.log(logMultipleIds('text', 'text')); // 4

**Literal Types**

// типизируем функцию для фетч запроса

// если запишем так, то сможем передать любое слово в method, которое и не является method

// метод должен быть get/post и т.д.

// поэтому создадим enum

enum RequestType {

  GET = 'get',

  POST = 'post',

}

function fetchWithAuth(Utl: string, method: RequestType) {

}

**Но делать такое на каждый шаг иногда лишняя работа.**

В TS есть возможность делать Literal Types.

В итоге мы можем передать или строку post, или строку get.

// сделаем через Literal Types

function fetchWithAuth(Utl: string, method: 'post' | 'get') {}

fetchWithAuth('s', 'post');

fetchWithAuth('s', 's'); // это будет неккоректно

// можем указать возвращаемый тип

// и тогда мы можем вернуть 1/-1

function fetchWithAuthAll(url: string, method: 'post' | 'get'): 1 | -1 {

  return 1;

}

let method = 'post'

// так не получится

fetchWithAuthAll('s', method);

const method = 'post';

И как вариант мы можем кастомнуть к определенному типу:

let method = 'post';

// const method = 'post';

// так не получится

// идет проверка на типы

// сделаем константой

fetchWithAuthAll('s', method as 'post');