Передовые инженерные школы

МИНОБРНАУКИ РОССИИ

УНИВЕРСИТЕТ ИННОПОЛИС

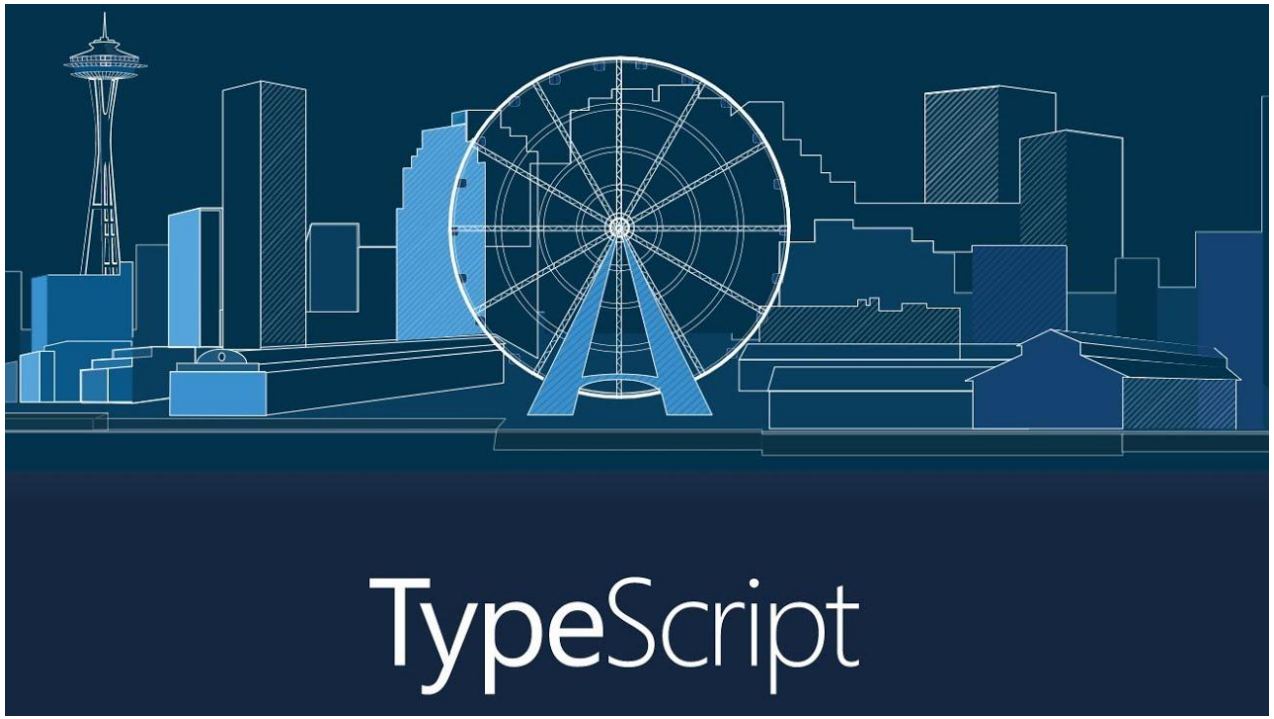
Занятие 6

TypeScript. Примитивные типы. Сложные типы. Дженерики

План занятия

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

1. TypeScript это и зачем?
2. Примитивные типы
3. Сложные типы
4. Литералы
5. Дженерики
6. Enums или перечисления



TypeScript это и зачем

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

TypeScript – язык разработанный Microsoft в 2012г. Средство разработки веб-приложений, бекэнд и десктоп. Расширяет возможности JavaScript

Плюсы TypeScript:

- помогает обнаруживать самые частые ошибки, на стадии написания кода
- повышает читаемость кода и поддержку контракта между фронт и бек –энд разработкой
- заставляет писать и понимать, что пишешь

Минусы TypeScript:

- написание кода занимает больше времени, т.к. нужна типизация
- typescript нужно компилировать

Упражнения для практики - https://typescript-exercises.github.io/

TypeScript это и зачем

Установка TypeScript глобально

npm i -g typescript

Проверить что все установилось

tsc -v

TypeScript это и зачем

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Компиляция TypeScript

1. Создаем **main.ts**
2. Пишем код в нем:
   let day = ‘Пятница’
   const CountWeek = 7
3. Компилируем **tsc main**

Если нужно следить за изменением файлов добавляем ключ **-w**
tsc main.ts –w

Во время компиляции если будут найдены ошибки, их выведет на экран

TypeScript это и зачем

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

Настройка TypeScript

Создаем файл конфигурации
tsc -init

В корне проекта появится файл **tsconfig.json**

Если указываеть входные файлы в коммандной строке (например, tsc main), файл tsconfig.json будет игнорироваться

```
{
    "compilerOptions": {
        ...
        /* Модули */
        "target": "es2016", // Измените на "ES2015", чтобы скомпилировать в ES6
        "rootDir": "./src", // Откуда компилировать
        "outDir": "./public", // Куда компилировать (обычно папка, которая разворачивается
на сервере)
        /* Поддержка JavaScript */
        "allowJs": true, // Позволяет компилировать JS-файлы
        "checkJs": true, // Проверяет типы в JavaScript-файлах и сообщает об ошибках

        /* Emit */
        "sourceMap": true, // Создать source map файлы для готовых файлов JavaScript
(хороошо подходит для дебаггинга)
        "removeComments": true, // Игнорировать комментарии
    },

    "include": ["src"] // Компилировать только файлы из папки src
}
```

Примитивные типы TypeScript

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Всего есть 7 примитивных типов:

- string
- number
- bigint
- boolean
- undefined
- null
- Symbol

В TypeScipt указывается нужный тип с помощью **:type**

```
let id: number = 5;
let firstname: string = 'Stalker';
let hasDog: boolean = true;
let unit: number; // Объявление переменной без присваивания значения
unit = 5;
```

Примитивные типы TypeScript

В большинстве случаев, лучше не указывать тип явно, так как TypeScript автоматически присваивает тип переменной (вывод типа)

```
let id = 5; // TS знает, что это число
let firstname = 'danny'; // TS знает, что это строка
let hasDog = true; // TS знает, что это логическое значение
hasDog = 'yes'; // ERROR - TS выдаст ошибку
```

Примитивные типы TypeScript

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Объединенный тип (union) - это переменная, которой можно присвоить более одного типа

```
let age: string | number;
age = 26;
age = '26';
```

Сложные типы TypeScript

Массив и его тип данных

```
let ids: number[] = [1, 2, 3, 4, 5]; // может содержать только цифры

let names: string[] = ['Денис', 'Аня', 'Богдан']; // может содержать только строки

let options: boolean[] = [true, false, false]; // может содержать true или false

let books: object[] = [
  { name: 'Алиса в Стране чудес', author: 'Льюис Кэррол' },
  { name: 'Идиот', author: 'Федор Достоевский' },
]; // может содержать только объекты

let arr: any[] = ['привет', 1, true]; // превращает TypeScript в JavaScript (об этом
чуть позже)

ids.push(6);
ids.push('7'); // ОШИБКА: Аргумент типа 'string' не может быть присвоен параметру
типа 'number'
```

Сложные типы TypeScript

Объединенные типы для объявления массива, который содержит в себе несколько типов данных:

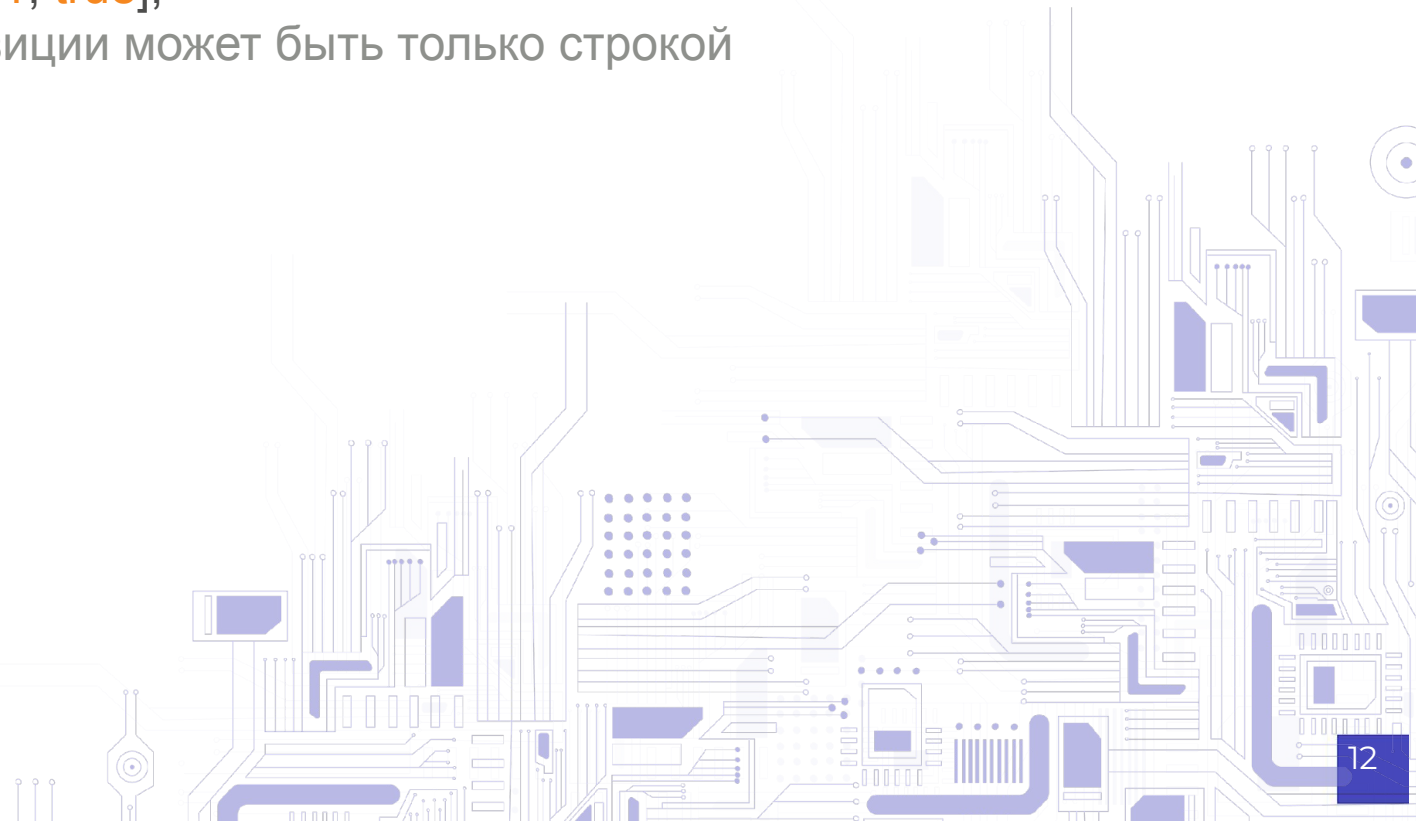
```
let person: (string | number | boolean)[] = ['Денис', 1, true];
person[0] = 100;
person[1] = {name: 'Денис'} // Ошибка - массив person не может содержать в себе объекты
```

Сложные типы TypeScript

В TypeScript можно объявить специальный тип массива – кортеж

Кортеж - это массив с фиксированным размером и известным набором данных
Он более строгий, чем обычные массивы

```
let person: [string, number, boolean] = ['Денис', 1, true];
person[0] = 100; // Ошибка - значение на 0 позиции может быть только строкой
```



Сложные типы TypeScript

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

Объекты в TypeScript должны содержать все объявленные свойства с теми же типами, которые были объявлены:

```
// Объявляем переменную-объект person со специальной аннотацией типов
let person: {
  name: string;
  location: string;
  isProgrammer: boolean;
};
// Присваиваем переменной person объект со всеми необходимыми полями и значениями
person = {
  name: 'Денис',
  location: 'RU',
  isProgrammer: true,
};
person.isProgrammer = 'Да'; // ОШИБКА: должно быть логическое значение

person = {
  name: 'Олег',
  location: 'RU',
};
// ОШИБКА: пропущено свойство isProgrammer
```

Сложные типы TypeScript

Для описания “шаблона” объекта можно использовать interface

```
interface Person {
  name: string;
  location: string;
  isProgrammer: boolean;
}

let person1: Person = {
  name: 'Денис',
  location: 'RU',
  isProgrammer: true,
};

let person2: Person = {
  name: 'Саша',
  location: 'Россия',
  isProgrammer: false,
};
```

Сложные типы TypeScript

С функциями в объекте

```
interface Speech {
  sayHi(name: string): string;
  sayBye: (name: string) => string;
}

let sayStuff: Speech = {
  sayHi: function (name: string) {
    return Привет, ${name};
  },
  sayBye: (name: string) => Пока, ${name},
};

console.log(sayStuff.sayHi('Питер')); // Привет, Питер
console.log(sayStuff.sayBye('Питер')); // Пока, Питер
```

Сложные типы TypeScript

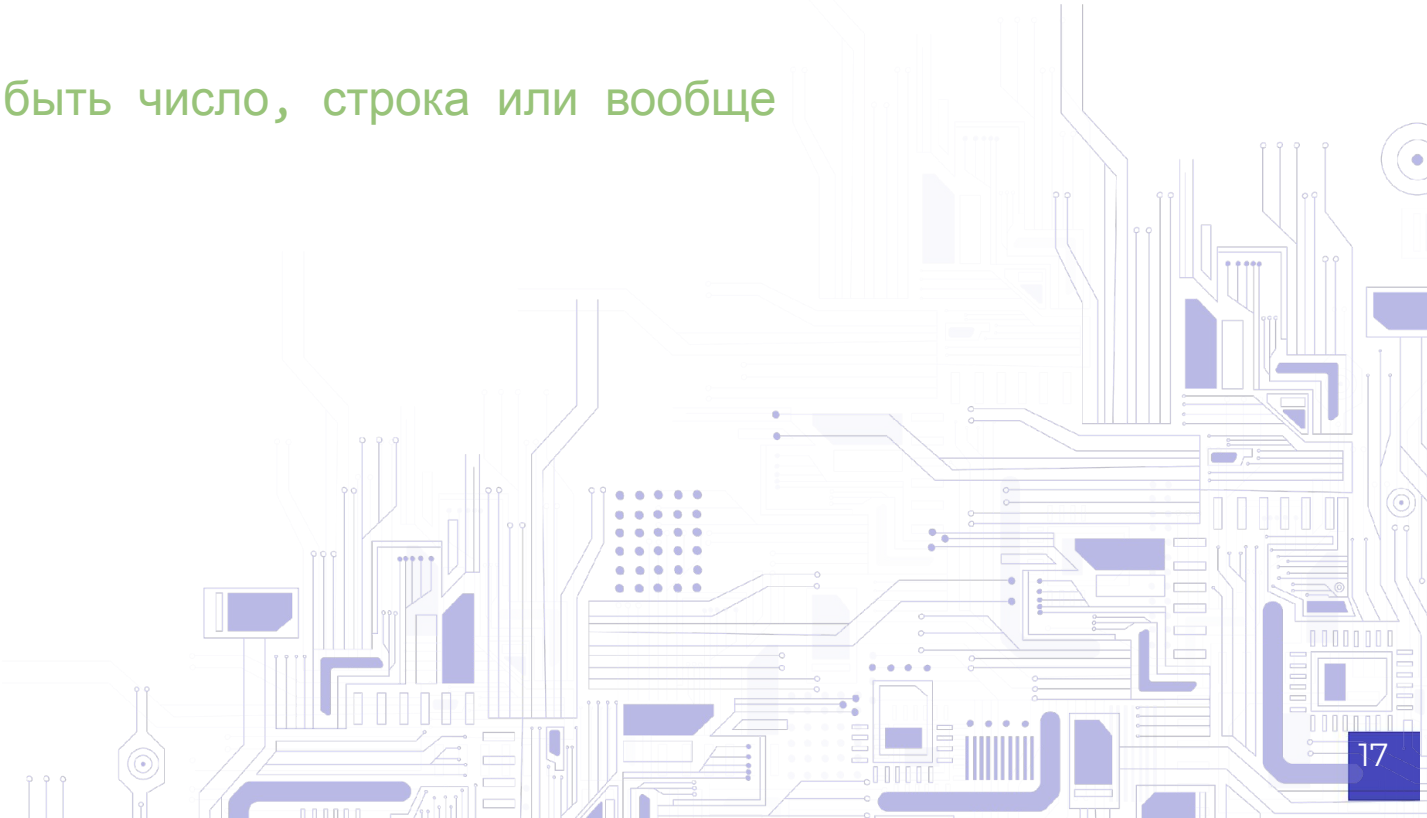
Функции в TypeScript

```
// Объявляем функцию circle, которая будет принимать числовую
переменную diam и возвращать строку
function circle(diam: number): string {
    return 'Длина окружности: ' + Math.PI * diam;
}

console.log(circle(10));

// В стрелке
const circle = (diam: number): string => {
    return 'Длина окружности: ' + Math.PI * diam;
}

console.log(circle(10));
```

Сложные типы TypeScript

Вопросительный знак после параметра, чтобы сделать его необязательным

```
const add = (a: number, b: number, c?: number | string) => {
    console.log(c);
  return a + b;
};

console.log(add(5, 4, 'Здесь могло бы быть число, строка или вообще
ничего!'));
```



Сложные типы TypeScript

Функция ничего не возвращает используется ключевое слово **void**

```
const logMessage = (msg: string): void => {
    console.log('Логи: ' + msg);
};

logMessage('Typescript - это круто!');
```

Сложные типы TypeScript

Динамические типы (any)

Используя тип any , мы можем превратить TypeScript обратно в JavaScript:

```
let age: any = '100';
age = 100;
age = {
    years: 100,
    months: 2,
};
```

Сложные типы TypeScript

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Псевдонимы типов

```
type StringOrNumber = string | number;
type PersonObject = {
  name: string;
  id: StringOrNumber;
};
const person1: PersonObject = {
  name: 'Федор',
  id: 1,
};
const person2: PersonObject = {
  name: 'Олег',
  id: 2,
};
const sayHello  = (person: PersonObject) => {
  return 'Привет, ' + person.name;
};
const sayGoodbye = (person: PersonObject) => {
  return 'Пока, ' + person.name;
};
```

Сложные типы TypeScript

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

TypeScript не имеет доступа к DOM. Это означает, что при обращении к DOM-элементам TypeScript не может быть уверен в том, что они существуют.

С оператором ненулевого подтверждения ! говорим компилятору, что выражение не равно **null** или **undefined**

Это может быть полезным, когда компилятор не может узнать, какой тип используется, но мы знаем это.

```
// Здесь мы говорим TypeScript, что эта ссылка существует
const link = document.querySelector('a')!;
console.log(link.href);
```

не нужно объявлять тип переменной link. TypeScript сам понимает (с помощью определения типа), что эта переменная типа HTMLAnchorElement

Сложные типы TypeScript

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Приведение типов (ключевое слово as):

```
const form = document.getElementById('signup-form') as HTMLFormElement;
console.log(form.method);
```

TypeScript имеет встроенный объект Event

```
const form = document.getElementById('signup-form') as HTMLFormElement;

form.addEventListent('submit', (e: Event) => {
  console.log(e.tarrget);  // ОШИБКА: Свойство 'tarrget' не существует у типа 'Event'. Может, вы имели ввиду 'target'?
});
```

Сложные типы TypeScript

Классы в TypeScript и модификатор доступа

```
class Person {
    readonly name: string; // это свойство неизменно - его можно только прочитать
    private isCool: boolean; // можно прочитать и изменять только в пределах этого класса
    protected email: string; // можно прочитать и изменить только из класса и наследуемых от него
    public friends: number; // можно прочитать и изменить откуда угодно, даже вне класса

    constructor(n: string, c: boolean, e: string, f: number) {
        this.name = n;
        this.isCool = c;
        this.email = e;
        this.friends = f;
    }

    sayMyName() {
        console.log(`Ты не Хайзенберг, ты ${this.name}`);
    }
}

const person1 = new Person('Менделеев', false, 'men@de.ru', 118);
console.log(person.name); // все в порядке
person1.name = 'Хайзенберг'; // ОШИБКА: только для чтения
console.log(person1.isCool); // ОШИБКА: private свойство - доступ есть только в пределах класса
Person
console.log(person1.email); // ОШИБКА: protected свойство - доступ есть только в пределах класса
Person и его наследниках
console.log(person1.friends); // public свойство - никаких проблем
```

Сложные типы TypeScript

Интерфейсы объявляются как объекты и выглядят

```
interface Person {
  name: string;
  age: number;
}

function sayHi(person: Person) {
  console.log(Привет, ${person.name});
}

sayHi({
  name: 'Джон',
  age: 33,
}); // Привет, Джон
```

Сложные типы TypeScript

Можно объявлять их как объекты, используя

```
type Person = {
    name: string;
    age: number;
}

function sayHi(person: Person) {
    console.log(Привет, ${person.name});
}

sayHi({
    name: 'Джон',
    age: 33,
}); // Привет, Джон
```

Сложные типы TypeScript

Тип объекта может быть указан анонимно, прямо в параметрах функции:

```
function sayHi(person: { name: string; age: number }) {
    console.log(`Привет, ${person.name}`);
}

sayHi({
    name: 'Джон',
    age: 33,
}); // Привет, Джон
```

Сложные типы TypeScript

Ключевое различие - псевдонимы нельзя наследовать, а интерфейсы можно.

Наследование интерфейса:

```
interface Animal {
    name: string
}

interface Bear extends Animal {
  honey: boolean
}

const bear: Bear = {
  name: 'Винни',
  honey: false,
}
```

типы не могут изменены после объявления:

```
type Animal = {
  name: string
}

type Animal = {
  tail: boolean
}
// ОШИБКА: Дублирующийся идентификатор
'Animal'.
```

Документация TypeScript советует использовать интерфейсы для объявления объектов, если не требуется использовать возможности типов

Сложные типы TypeScript

Расширение типов

```ts
type Animal = {
  name: string
}

type Bear = Animal & {
  honey: boolean
}

const bear: Bear = {
  name: 'Винни',
  honey: true,
}
```

Сложные типы TypeScript

Добавление новых полей к существующему интерфейсу:

```
interface Animal {
  name: string
}

// Добавление поля к интерфейсу
interface Animal {
  tail: boolean
}

const dog: Animal = {
  name: 'Хатико',
  tail: true,
}
```

Литералы

Вместо общих основных типов string и number можно указывать конкретные строки и числа

```
// Объединенный тип с литералами
let favoriteColor = 'red' | 'blue' | 'green' | 'yellow';

favoriteColor = 'blue';
favoriteColor 'black'; // ОШИБКА: Тип "'black'" не может быть
присвоен типу "'red' | 'blue' | 'green' | 'yellow'"
```

Дженерики (Generics)

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Дженерики штуки, которые дают возможность создавать компоненты, которые могут работать с несколькими типами, а не привязаны только к одному типу.

Дженерики описывают более универсальный и переиспользуемый компонент

Дженерики (Generics)

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

```
const addId = (obj: object) => {
    let id = Math.floor(Math.random() * 1000);

    return { ...obj, id };
};

let person1 = addId({ name: 'Джон', age: 40 });

console.log(person1.id); // 271
console.log(person1.name): // ОШИБКА: Свойство 'name' не
существует для типа '{ id: number; }'.
```

Дженерики (Generics)

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Добавляем дженерик, обозначается как <T>, где T – это параметр типа

```
// <T> используется для примера, мы можем использовать любую букву,
например <X> или <A>
const addId = <T>(obj:T) => {
    let id = Math.floor(Math.random() * 1000);

    return { ...obj, id };
};
```

Дженерики (Generics)

Передаем объект в addId, и говорим TypeScript установить тип как в T, который может стать любым типом.

addId теперь будет знать, какие свойства имеет объект, который передаем в эту функцию.

Однако можно передать в addId все что угодно, и TypeScript примет это, не выводя никаких ошибок:

```
let person1 = addId({ name: 'Джон', age: 40 });
let person2 = addId('Салли'); // Передаем строку и никакой ошибки

console.log(person1.id); // 271
console.log(person1.name): // Джон

console.log(person2.id); // 890
console.log(person2.name); // ОШИБКА: Свойство 'name' не
существует для типа '"Салли" & { id: number; }'.
```

Дженерики (Generics)

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

TypeScript не видит проблемы
Он сообщает об ошибке только когда пытаемся обратиться к свойству name.

Необходимо сказать TypeScript, что можно передавать только объекты.
Для этого надо сделать наш дженерик T расширением object:

```
const addId = <T extends object>(obj:T) => {
    let id = Math.floor(Math.random() * 1000);

    return { ...obj, id };
};

let person1 = addId({ name: 'Джон', age: 40 });
let person2 = addId('Салли'); // ОШИБКА: Невозможно задать
аргумент типа 'string' для параметра типа 'object'.
```

Дженерики (Generics)

Сейчас получаем снова ошибку:

```
let person2 = addId(['Салли', 26]); // Передаем в функцию массив - никаких проблем

console.log(person2.id); // 890
console.log(person2.name); // ОШИБКА: Свойство 'name' не существует для типа '(string | number)[] & { id: number; }'.
```

Дженерики (Generics)

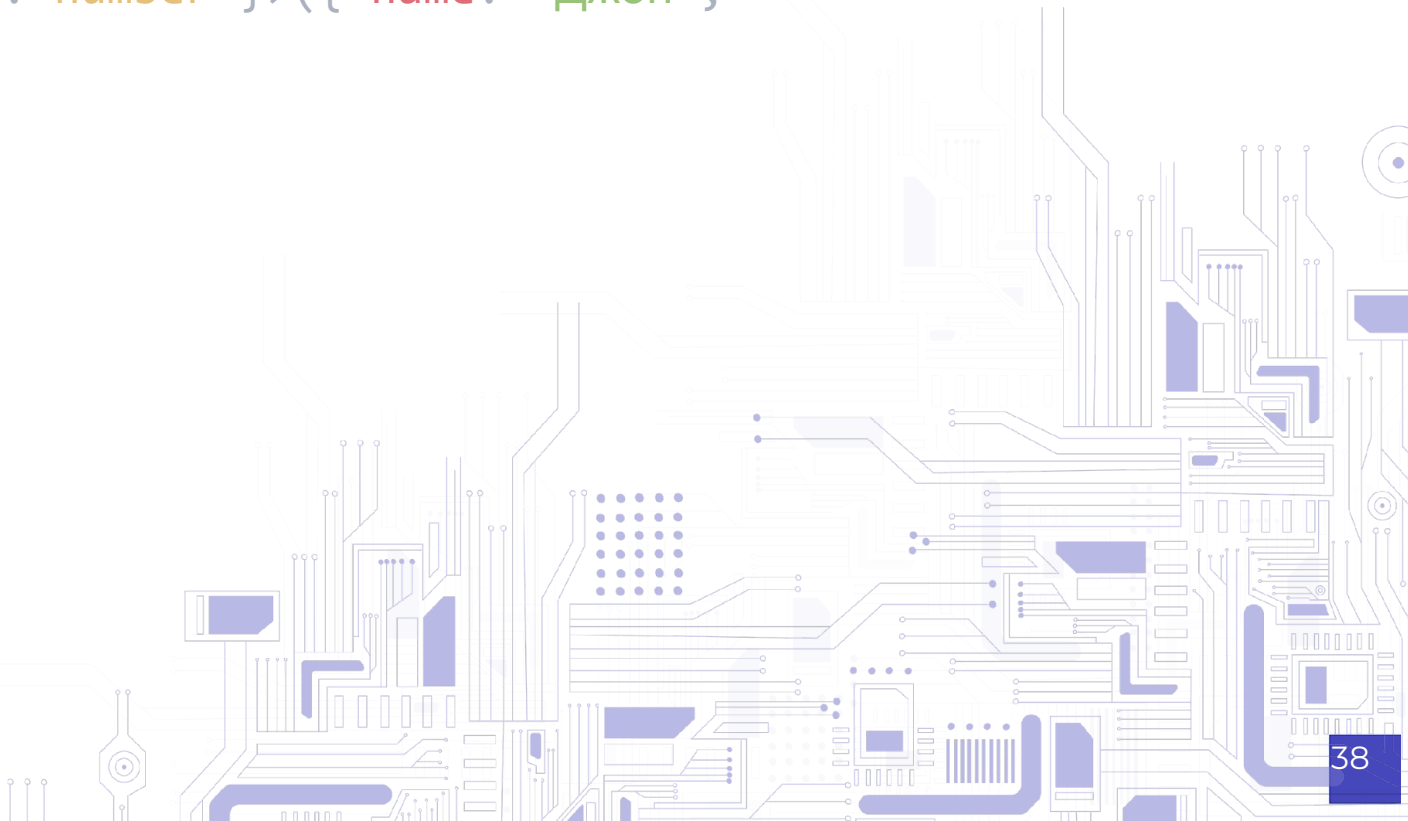
Еще исключим для точности массивы

```
const addId = <T extends { name: string }>(obj: T) => {
    let id = Math.floor(Math.random() * 1000);

    return { ...obj, id };
};
let person2 = addId(['Салли', 26]); // ОШИБКА: Аргумент должен
содержать свойство 'name' со строковым значением.
```

Дженерики (Generics)

Тип можно передать в угловых скобках при вызове функции, однако это не обязательно, так как TypeScript сам проверяет это.

```
// явно указываем между угловыми скобками, какого типа должен быть аргумент
let person1 = addId<{ name: string; age: number }>({ name: 'Джон', age: 40 });
```



Дженерики (Generics)

Почему дженерики лучше any

```
function logLength(a: any) {
    console.log(a.length); // Все в порядке
    return a;
}

let hello = 'Hello world';
loglength(hello); // 11

let howMny = 8;
logLength(howMany); // undefined (TypeScript не показывает ошибку, а
хотелось бы)
```

Дженерики (Generics)

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Прикрутим дженерик <T>

```
function logLength<T>(a: T) {
    console.log(a.length); // ОШИБКА: TypeScript не уверен, что
'a' имеет свойство 'length'
    return a;
}
```

Дженерики (Generics)

А теперь с использованием наследования интерфейса

```
interface hasLength {
    length: number;
}

function logLength<T extends hasLength>(a: T) {
    console.log(a.length); // Теперь ошибки нет
    return a;
}

let hello = 'Hello world';
loglength(hello); // 11

let howMny = 8;
logLength(howMany); // ОШИБКА: у числа нет свойства 'length'
```

Дженерики (Generics)

С интерфейсом и массивом элементов

```
interface hasLength {
    length: number;
}
function logLengths<T extends hasLength>(a: T[]) {
    a.forEach(element => {
        console.log(element.length);
    });
}
let arr = [
    'У этой строки есть длина',
    ['У массива', 'тоже есть', 'длина'],
    { material: 'plastic', length: 40 },
];
logLengths(arr);
```

все элементы будут имеют свойство length

Дженерики (Generics)

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Дженерики с интерфейсами
При создании бывает так что не знаем, какой конкретно тип определенного объекта будет передаваться в будущем

```
// Тип T будет передан в интерфейс
interface Person<T> {
    name: string;
    age: number;
    documents: T;
}
// тип поля 'documents', в данном случае - массив строк
const person1: Person<string[]> = {
    name: 'Павел',
    age: 21,
    documents: ['паспорт', 'полис', 'снилс'],
};
// тип 'string'
const person1: Person<string> = {
    name: 'Павел',
    age: 21,
    documents: 'паспорт, зачетка',
};
```

Enums или перечисления

Объявляется коллекция связанных значений (строк или чисел) в виде набора именованных констант

```
enum ResourceType {
    BOOK,
    AUTHOR,
    FILM,
    DIRECTOR,
    PERSON,
}
console.log(ResourceType.BOOK); // 0
console.log(ResourceType.AUTHOR); // 1
// Начать с 1
enum ResourceType {
    BOOK = 1,
    AUTHOR,
    FILM,
    DIRECTOR,
    PERSON,
}
console.log(ResourceType.BOOK); // 1
console.log(ResourceType.AUTHOR); // 2
```

Enums или перечисления

Enums основаны на числах и хранят строковые значения, в виде чисел. Однако могут быть строками:

```
enum Direction {
    Up = 'Up',
    Right = 'Right',
    Down = 'Down',
    Left = 'Left',
}

console.log(Direction.Right); // Right
console.log(Direction.Down); // Down
```

Передовые инженерные школы

МИНОБРНАУКИ РОССИИ

УНИВЕРСИТЕТ ИННОПОЛИС

Спасибо за внимание