NAUCZ SIĘ TYPESCRIPT W 3 GODZINY!

CZEŚĆ!

Tomasz Nastały

JavaScript Developer @ 7N

Angular Tricity

angular.love

FeFAQ

infoShare academy

AGENDA

1. TypeScript - nieco teorii
2. Obiektowość w TS
3. Typowanie
4. Enums
5. Kompatybilność typów
6. Typowanie zaawansowane
7. Dekoratory
8. Plik tsconfig.json

~19.30 – 19.40

WIFI

1. SIEĆ: **HighSpeed** / HASŁO: **O4forever**
2. SIEĆ: **O4** / HASŁO: **O4forever**

REPO DO ZADAŃ

1. git clone https://github.com/tomasznastaly/isa-warsztat-ts.git
2. npm install / yarn install

DLA LENIWYCH ;)

Branche z rozwiązanymi zadaniami:

isa-ts/zad-1-obiektowosc

isa-ts/zad-2-typowanie

isa-ts/zad-3-enums

isa-ts/zad-4-zaawansowane-typy

isa-ts/zad-5-dekoratory

isa-ts/develop (całość)

OCZEKIWANY EFEKT

https://tomasznastaly.github.io/marvel

1. TypeScript – nieco teorii

TypeScript – co to jest?

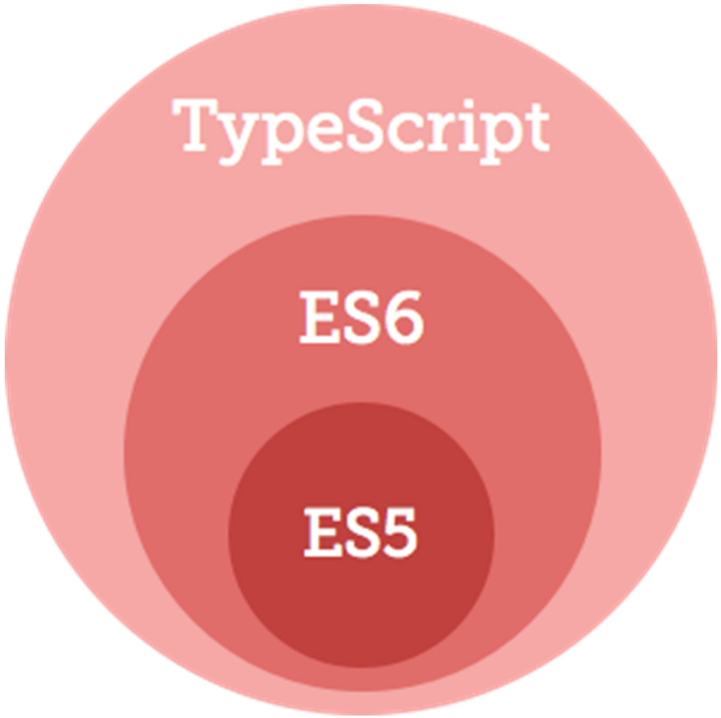
TypeScript – opcjonalnie typowany nadzbiór języka JavaScript, kompilowany do JavaScriptu.

Stworzony przez firmę Microsoft i rozwijany od 2012 roku, obecnie najnowsza wersja to 2.8 (maj 2018).

„TypeScript is a syntactic sugar for JavaScript. TypeScript syntax is a superset of ECMAScript 2015 (ES2015) syntax. Every JavaScript program is also a TypeScript program”

SPECYFIKACJA: https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md

TypeScript & ES6



TypeScript – features

- Statyczne typowanie
- Interfejsy
- klasy abstrakcyjne
- enums
- tworzenie pól i metod prywatnych
- dekoratory

TypeScript – zastosowanie

- Dowolny kod JS
- NodeJS
- **Angular (domyślnie)**
- React

TypeScript – korzyści

- Jeśli preferujemy OOP, to TypeScript jest dużo wygodniejszy niż czysty JS
- Wyłapywanie błędów w kodzie na etapie pisania dzięki typom (literówki!)
- Łatwiejszy kod do zrozumienia dla ludzi kodujących po stronie backendu (.NET, Java np.)
- Łatwiejszy start w frameworkach, np. Angular
- Dzięki znajomości TS, łatwiej przeskoczyć do języków backendowych
- Ciągle rozwijany w kolejnych wersjach
- Lepsza czytelność kodu
- Elastyczność! TS nic nie narzuca

TypeScript – jak zacząć?

- https://www.typescriptlang.org/samples/

1. **npm install –g typescript**
2. stworzyć plik z rozszerzeniem .ts
3. wpisać w terminalu „**tsc nazwa_pliku.ts**”

Profit: pojawia się skompilowany plik .js

TIP: lepiej korzystać z dobrego starter-kita opartego o bundler (np. Webpack):

https://github.com/emyann/typescript-webpack-starter

2. Obiektowość w TS

Modyfikatory dostępu

Modyfikatory dostępu – wpływają na widoczność elementu, który poprzedzają (np. na pole klasy).

Modyfikatory w TypeScript:

- public (domyślny)
- private
- protected
- readonly

Modyfikator public

Public – pole lub metoda są widoczne spoza klasy. W TypeScript, public jest domyślnym modyfikatorem. Nie musimy go pisać.

```
class Hero {
    name; // public name;

    constructor(name) {
        this.name = name;
    }
}
```

```
const hulk = new Hero('Hulk’);

hulk.name; // 'Hulk’

hulk.name = 'Ironman';
```

Modyfikator private

Private – pole lub metoda są niewidoczne spoza klasy.

```
class Hero {
    private name;

    constructor(name) {
        this.name = name;
    }
}
```

```
const hulk = new Hero('Hulk’);

hulk.name; // ERROR!
hulk.name = 'Ironman’; // ERROR!
```

Modyfikator protected

Protected – zachowuje się podobnie jak **private**, ale klasa pochodna ma dostęp.

KLASA BAZOWA

```
class Hero {
    protected speed = 60;
    ...
}
```

KLASA POCHODNA

```
class WeakHero extends Hero {
    ...

    logSpeed() {
        console.log(this.speed);
    }
}
```

Modyfikator readonly

Readonly – pole wyłącznie do odczytu, modyfikator niedostępny dla metod. Do pola readonly można przypisać wartość wyłącznie podczas inicjalizacji lub w konstruktorze.

```
class Hero {
    name;
    readonly speed = 100;

    constructor(name, speed) {
        this.name = name;
        this.speed = speed;
    }

    setSpeed(speed) {
        this.speed = speed;
    }
}
```

KLASA POCHODNA

```
const hulk = new Hero('Hulk', 400);

hulk.speed; // 400

hulk.speed = 500 // ERROR! TYLKO DO ODCZYTU
```

Klasa abstrakcyjna

- nie można stworzyć instancji klasy abstrakcyjnej, jak nazwa wskazuje, jest to coś abstrakcyjnego
- pozwala tworzyć bardziej elastyczne struktury
- pozwala unikać powtarzalności kodu
- metody klasy abstrakcyjnej mogą posiadać implementację

```
abstract class Hero {
    strength = 100;
}

const hulk = new Hero(); // ERROR! NIE MOŻNA STWORZYĆ INSTANCJI
KLASY ABSTRAKCYJNEJ
```

Metoda abstrakcyjna

- Nie może posiadać implementacji
- Musi zostać zaimplementowana w klasie pochodnej

```
abstract class Hero {
    strength = 100;
    abstract kick();
}
```

```
class WeakHero extends Hero {
    kick() {
        return this.strength * 2;
    }
}
```

Pole statyczne, metoda statyczna

- pole statyczne jest właściwością klasy
- w polu statycznym nie może wystąpić this (this w klasie wskazuje na obiekt stworzony na podstawie tej klasy (instancję))
- „under the hood” w TypeScript, pole statyczne jest deklarowane jako property function constructora
- aby uzyskać dostęp do pola lub metody statycznej, uprzedzamy ją nazwą klasy

```
class Hero {
    static nextId = 1;
    id;

    constructor() {
        this.id = Hero.nextId++;
    }
}
```

```
const hulk = new Hero();
const ironman = new Hero();

hulk.id // 1;
ironman.id // 2;
Hero.nextId // 3;
```

Pole statyczne, metoda statyczna

- Człony statyczne i dotyczące instancji żyją w osobnych „declaration space”, stąd poniższy zapis jest możliwy

```
class Hero {
    id: number = 10; // Instance member
    static id: string; // Static member
}
```

Gettery / Settery

- Dobra enkapsulacja gwarantuje, że jedynym obiektem odpowiedzialnym za zmianę stanu jest sam obiekt
- TypeScript udostępnia nam gettery i settery poprzez słowa kluczowe **get** i **set**
- Niektórzy uważają gettery i settery za antywzorzec: decyzja należy do Ciebie

PROBLEM – MODYFIKACJA Z ZEWNĄTRZ

```
class Hero {
    id = 10;
}

const hulk = new Hero();
hulk.id = 20; // MOŻLIWE NADPISANIE;
```

GETTERY / SETTERY

```
class Hero {
    private _id = 10;

    get id() {
        return this._id;
    }

    set id(newId) { // OPCJONALNIE
        this._id = newId;
    }
}
const hulk = new Hero();
hulk.id // 10
hulk.id = 20; // OK
```

Super

- **super()** woła konstruktor klasy bazowej (musi być zawołany z tymi samymi parametrami)
- Musi być pierwszą instrukcją w konstruktorze klasy pochodnej
- Jeśli klasa pochodna ma konstruktor, to musi wołać super
- W metodach klasy pochodnej możemy sięgać po metody klasy bazowej poprzez super.nazwaMetody()

```
class Hero {
  constructor(public name) {}
}
```

```
class SuperHero extends Hero {

  constructor(name, public strength) {
    super(name);
  }

}
```

Shorthand w konstruktorze

```
class Hero {
    private name;

    constructor(name) {
        this.name = name;
    }
}
```

=

```
class Hero {
  constructor(private name) {}
}
```

Obiektowość w TS – DO KODU!

Obiektowość - zadanie

- Stwórz klasę **DOM** z metodą statyczną **renderHeroes**
- Stwórz klasę **abstrakcyjną BaseHero** oraz klasę **StrongHero**, która dziedziczy po klasie abstrakcyjnej **BaseHero**
- Stwórz klasę **HeroesService** z metodą **getHeroes**, która skonsumuje heroesów z REST API i zmapuje heroesów do **StrongHero**
- W **index.ts** wstrzyknij w konstruktor **HeroesService** i wyrenderuj pobranych **heroesów** za pomocą **DOM.renderHeroes(heroes)**
- Heroesi przed renderowaniem niech stoczą walkę

3. Typowanie

Typowanie

- typowanie w TS jest całkowicie opcjonalne
- typowanie w TS jest statyczne, czyli **typ zmiennej sprawdzany w czasie compile-time** a nie run-time, a typy nadajemy jawnie na danych zmiennych
- możemy tworzyć złożone typy poprzez interfejsy lub słowo kluczowe **type**
- TypeScript posiada wbudowaną inferencję typów
- typowanie pozwala nam uniknąć błędów już na etapie pisania samego kodu
- mniejsza ilość przypadków testowych

Dodanie typu

- Typ dodajemy za pomocą dwukropka

```
const name: string = 'John Doe’;

const names: string[] = ['John', 'Monica’];

class Hero {
    kick(): number {
        return 30;
    }
}

const names: Array<string> = ['John', 'Monica'];
```

Typy podstawowe

- **number** (age: number)
- **string** (name: string)
- **boolean** (isDrunk: boolean)
- **null** (validator(): ErrorObj | null)
- **undefined** (validator(): ErrorObj | undefined)
- **array** (names: string[])
- **tuple** ([string, number, boolean])
- **enum**
- **void** (log() { console.log() })
- **never** (throw Error(), while(true))

Inferencja typów

- Inferencja typów polega na tym, że TypeScript sam wnioskuje typ na podstawie przypisanej lub zwracanej wartości
- Warto korzystać z inferencji typów, zmniejsza narzut typowania w kodzie

```
const name: string = 'John Doe';

const names: string[] = ['John', 'Monica'];

const isOverEighteen = (age): boolean =>  {
    return age > 18;
};
```

Tworzenie interfejsu

- Do stworzenia złożonych typów, możemy wykorzystać interfejsy
- W przeciwieństwie do klas, interfejsy są usuwane z kodu wynikowego

```
interface Person {
    name: string;
    age: number;
    isAdult: boolean;
    address: {
        street: string;
        city: string;
        postal: string;
    }
}
```

```
interface Person {
    name: string;
    age: number;
    isAdult: boolean;
    address: Address;
}

interface Address {
    street: string;
    city: string;
    postal: string;
}
```

```
fetch(url)
    .then((persons: Person[]) => console.log(persons));
```

Interface vs Type

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

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

- Interfejsy można rozszerzać (z użyciem słowa kluczowego extends)
- W przeciwieństwie do Type, deklaracje interfejsów są łączone

```
interface Person {
    name: string;
}

interface Person {
    age: number;
}
```

Rzutowanie (type casting)

```
let foo = {}; // TS ifneruje, że to pusty obiekt
foo.bar = 123; // ERROR: Property 'bar' does not exist on type '{}'
foo.bas = 'hello'; // ERROR: Property 'bas' does not exist on type '{}'
```

```
interface Foo {
  bar: number;
  bas: string;
}
```

TIP dla Reaktowców: w rzutowaniu w plikach **.tsx,** <type> jest niedostępny (konflikt), trzeba używać „as” (W Angularze oczywiście można, to poważny framework)

```
let foo = {} as Foo; (LUB let foo = <Foo>{};)

foo.bar = 123;
foo.bas = 'hello';
```

Typowanie – DO KODU!

Typowanie - zadanie

- Stwórz interfejs **Hero** i **AttackingHero**
- **StrongHero** niech implementuje interfejs **AttackingHero**
- **BaseHero** niech implementuje interfejs **Hero**
- Otypuj miejsca gdzie TS nie skorzysta z inferencji typów
- Stwórz typy reprezentujące odpowiedź z serwera

4. Enums

Enums

- **Enums** są przydatne szczególnie w miejscach, gdzie zbiór możliwych wartości jest ograniczony (np. dni tygodnia, kierunki, status zamówienia)
- W TS występują **enums** i **string enums**

```
enum Direction {
    Up, // 0
    Down, // 1
    Left, // 2
    Right, // 3
}
```

PO

KOMPILACJI

```
▼ Object i
    0: "Up"
    1: "Down"
    2: "Left"
    3: "Right"
    Down: 1
    Left: 2
    Right: 3
    Up: 0
```

String Enums

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}
```

PO

KOMPILACJI

```
▼ Object i
    Down: "DOWN"
    Left: "LEFT"
    Right: "RIGHT"
    Up: "UP"
```

Enums – DO KODU!

Enums - zadanie

- Stwórz enum o nazwie **Status** z wartościami **ALIVE** oraz **DEAD**
- dodaj pole status o typie Status do interfejsu Hero
- Stwórz w abstrakcyjnej klasie **BaseHero** nowe pole, które trzyma **Status**.**ALIVE** lub **Status**.**DEAD** w zależności od Math.random()
- Jeśli hero jest martwy, to niech jego obrazek jest czarnobiały, skorzystaj z klasy „dead” na tagu img

5. Kompatybilność typów

Kompatybilność typów

Typowanie w TypeScript jest strukturalne.

Podstawową zasadą strukturalnego typowania w TS, jest to, że typ A jest kompatybilny z typem B, jeśli B posiada przynajmniej te same człony (members) co A.

```
interface GreatHero {
    name: string;
    strength: number;
}

let hulk: GreatHero;
let magicJohnson = { name: 'Magic', strength: 100, age: 50 };

hulk = magicJohnson;
```

To nic, że greatHero nie ma pola „age”, ważne, że magicJohnson ma wszystkie pole wymagane przez GreatHero

Kompatybilność typów – Object literal przypisany bezpośrednio

```
interface GreatHero {
    name: string;
    strength: number;
}

let hulk: GreatHero = {
    name: 'Magic',
    strength: 100,
    age: 50
};
```

ERROR:
Type '{ name: string; strength: number; age: number; }' is not assignable to type 'GreatHero'.

Object literal may only specify known properties, and 'age' does not exist in type 'GreatHero'.

Parametry funkcji

```
interface GreatHero {
    name: string;
    strength: number;
}
```

- Ważne, że wartość przekazana jako parametr, posiada przynajmniej wszystkie człony, które są wymagane

```
let magicJohnson = { name: 'Magic', strength: 100, age: 50 };

function heroSayHello(hero: GreatHero) {
    return `Hello, i am ${hero.name}`;
}
heroSayHello(magicJohnson);
```

Kompatybilność funkcji

```
let sayHello = (name: string) => 'Hello ' + name;
let sayGoodbye = (name: string, lastName: string) => 'Bye '+ name;

sayGoodbye = sayHello;
sayHello = sayGoodbye;
```

Kompatybilność enums

```
enum HeroStatus {
    Alive,
    Dead
}

enum PaymentStatus {
    Pending,
    Completed,
    Refunded
}
```

- Enums są kompatybilne z liczbami, ale Enum nie jest kompatybilny z innym Enumem

```
let hulkStatus = HeroStatus.Alive;

hulkStatus = PaymentStatus.Pending;

const paymentStatus: PaymentStatus = 30;

const ironmanStatus: HeroStatus = 40;
```

Kompatybilność klas

```
class Batman {
    kick(input: number): number {
        return 5;
    }
}

class Superman {
    kick(power: number): number {
        return 3;
    }
}

const clarkKent: Batman = new Superman();
```

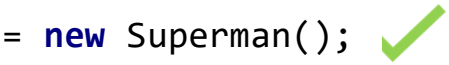
- Dwie klasy są kompatybilne, jeśli posiadają tą samą strukturę

Kompatybilność klas – pola statyczne

```
class Batman {
    static kills = 100;
    static kill() {};

    kick(input: number): number {
        return 5;
    }
}

class Superman {
    kick(power: number): number {
        return 3;
    }
}
const clarkKent: Batman = new Superman();
```



- Pola statyczne nie wpływają na kompatybilność typów

Kompatybilność klas – konstruktory

```
class Batman {
    constructor(city: string, endurance: number) {};
    kick(input: number): number {
        return 5;
    }
}
class Superman {
    constructor(speed: number) {};
    kick(power: number): number {
        return 3;
    }
}
const clarkKent: Batman = new Superman(80);
```

- Konstruktory nie wpływają na kompatybilność typów, o ile nie dodajemy w nich modyfikatorów

6. Typowanie zaawansowane

Union Type: |

- Union Type (|) pozwala, aby wartość mogła być danego typu LUB innego
- W użyciu wraz z „type”, można wymusić, jaką dokładnie wartość może przyjąć string

```
const calculateWidth = (width: string | number) => {};

type Easing = "ease-in" | "ease-out" | "ease-in-out";

class UIElement {
    animate(x: number, y: number, easing: Easing) {
    ...
    }
}
```

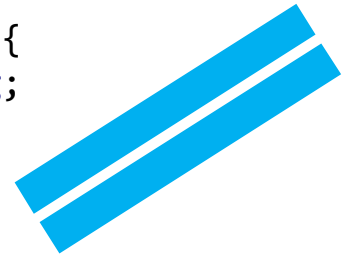
Intersection Type: &

- Intersection Type (&) składa parę typów w jeden

```
interface Product {
    productId: number;
    sentDate: string;
}

interface CancelableProduct {
    cancellationDate: string;
    paid: boolean;
}

class Shop {
    order(product: Product & CancelableProduct) {}
}
```



```
{
    productId: number;
    sentDate: string;
    cancellationDate: string;
    paid: boolean;
}
```

Sygnatura indeksu (Index Signature)

- Sygnatura indeksu przydaje się w sytuacjach, kiedy nie znamy dokładnych kluczy, ale znamy wartości, jakie przyjmą

```
interface Neighbor {
    area: string;
    population: number;
    hasMoreThanFourNeighbors: boolean;
}

interface PolishNeighborsInfo {
    [countryName: string]: Neighbor
}
```

Sygnatura indeksu - readonly

- Można użyć **readonly** wraz z sygnaturą, aby zapobiec możliwości przypisania wartości do indeksów

```
interface ReadonlyArrayOfStrings {
    readonly [index: number]: string;
}

let myArray: ReadonlyArrayOfStrings = ["Alice", "Bob"];
myArray[2] = "Mallory"; // error!
```

```
ERROR:
Index signature in type 'ReadonlyArrayOfStrings' only permits reading.
```

Enum i type – problem dodawania nowych kluczy

```
enum Technology {
    html = 'HTML',
    css = 'CSS',
    js = 'JS',
    angular = 'Angular',
    react = 'React',
    git = 'GIT',
    other = 'Inne'
}
```

```
type TechnologyKey = 'html' | 'css' |
'js' | 'angular' | 'react' | 'git' |
'other';
```

```
const getQuestions = (tech: TechnologyKey) => {
    console.log(tech);
};

getQuestions('html');

getQuestions('vue');
```

Rozwiązanie problemu: keyof + typeof

- keyof operuje na typach
- typeof bierze wartość i produkuje typ (zatem jak np. chce pobrać klucze interface, to typeof jest zbędne, bo interface to type a nie value)

```
enum Technology {
    html = 'HTML',
    css = 'CSS',
    js = 'JS',
    angular = 'Angular',
    react = 'React',
    git = 'GIT',
    other = 'Inne'
}
```

```
export type TechnologyKey = keyof typeof Technology;
```

```
const getQuestions = (tech: TechnologyKey) => {
    console.log(tech);
};

getQuestions('html');

getQuestions('vue');
```

Typy generyczne

PROBLEM: do kolejki wrzucam elementy jakie mi się podobają (any)

```
class Queue {
    private data: any[] = [];

    push(item: any) {
        this.data.push(item);
    }
    pop() {
        this.data.shift();
    }
}

const heroes = new Queue();
heroes.push('Hulk’);
heroes.push({ name: 'Ironman’ });
```

- W tej sytuacji, nie mamy żadnej korzyści z możliwości typowania. Do kolejki możemy wpychać co nam się podoba.

```
class Hero {
    name: string;
    power: number;
}
```

Typy generyczne

- Typy generyczne przydają się w sytuacjach, kiedy nasze skrypty operują na wartościach o różnych typach

```
class Queue<T> {
    private data: T[] = [];

    push(item: T) {
        this.data.push(item);
    }
    pop() {
        this.data.shift();
    }
}
```

```
const heroes = new Queue<Hero>();
heroes.push({ name: 'Ironman' });
```

```
interface Hero {
    name: string;
    power: number;
}
```

```
ERROR:
Argument of type '{ name: string; }' is not
assignable to parameter of type 'Hero'.

Property 'power' is missing in type '{ name:
string; }'
```

Typy generyczne

```
interface Hero { name: string; power: number; }
interface WeakHero { dead: boolean; name: string; power: number;}
interface SuperHero { immortal: boolean;}
```

```
class Queue<T extends Hero> {
    private data: T[] = [];
    push(item: T) { this.data.push(item); }
    pop() { this.data.shift(); }
}
```

```
const heroes = new Queue<WeakHero & SuperHero>();
heroes.push({ name: 'Hulk', power: 300, dead: true, immortal: false });
```

PRZYKŁAD: http://tiny.pl/g2l5t

Typowanie zaawansowane – DO KODU!

Typowanie zaawansowane - zadanie

- Korzystając z typów generycznych, otypuj metodę **attack** w interfejsie **AttackingHero** tak, aby móc walidować przekazany typ ataku (w przypadku przekazania jako parametru nieistniejącego ataku, powinniśmy dostać error)

7. Dekoratory *

Dekoratory

Dekoratory są funkcjami, które mogą modyfikować:

- klasy
- metody
- parametry
- pola klasy
- akcesory

PRZYKŁADY JAK PISAĆ:
https://netbasal.com/inspiration-for-custom-decorators-in-angular-95aeb87f072c
https://netbasal.com/automagically-unsubscribe-in-angular-4487e9853a88
https://gist.github.com/remojansen/16c661a7afd68e22ac6e
https://github.com/arolson101/typescript-decorators

Dekoratory & tsconfig.json

Uruchomienie dekoratorów w TS w pliku tsconfig.js:

- "experimentalDecorators": true

Decorator factory

```
function color(value: string) { // to jest decorator factory
    console.log('evaluating…');

    return function (target) { // to jest dekorator, target wskazuje
                                  na element który modyfikujemy

        console.log('calling...');
        // zrób coś z 'target' i 'value'...

    }
}
```

Aplikowanie dekoratorów

Dane są dekoratory: f() {}, g() {}

W jednej linii:

```
@f @g
methodA() {}
```

Wiele linii:

```
@f
@g
methodA() {}
```

```
f(): evaluated
g(): evaluated
g(): called
f(): called
```

Dekoratory – DO KODU!

Dekoratory - zadanie

- Stwórz dekorator **@Unenumerable** dla metody, który przestawia wartość enumerable w propertyDescriptor na false

8. Plik tsconfig.json

Plik tsconfig.json

- Plik tsconfig.json odpowiada za ustawienia kompilatora,

https://basarat.gitbooks.io/typescript/docs/project/tsconfig.html

- Tryb strict „true” wymusza typowanie i uruchamia tryb „strict”
- Domyślny target to ES3
- outFile to zło! https://basarat.gitbooks.io/typescript/docs/tips/outFile.html
- Oprócz compilerOptions możemy dodać exclude i include (jako tablice ścieżek)
- Tworzymy go komendą *„tsc --init”* (pod warunkiem, że TS jest zainstalowany globalnie)
- Obecność pliku tsconfig.json świadczy, że katalog w którym jest to root directory
- Może być ich wiele i mogą po sobie dziedziczyć

DZIĘKI ZA UWAGĘ!

Tomasz Nastały

nastalytomasz@gmail.com

Credits:
THX dla **Michał Michalczuk** za review