

# ZAAWANSOWANE PROGRAMOWANIE W TYPESCRIPT

Mateusz Jabłoński

---

mateusz.jablonski@sages.io

sages





# Mateusz Jabłoński

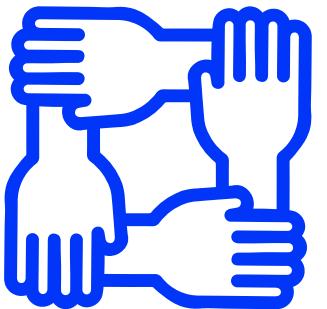
[mateuszjablonski.com](http://mateuszjablonski.com)

# Ustalenia

sages



Cel i agenda



wzajemne  
oczekiwania



Pytania i  
dyskusje



Elastyczność



Otwartość i  
uczciwość

# Agenda

Co nas czeka?

## > **Wprowadzenie do TypeScript**

Czym jest TypeScript? / Podstawy konfiguracji projektu / Omówienie komplikacji TypeScript do JavaScriptu i mapowanie kodu źródłowego / TypeScript vs JavaScript

## > **System typów w Typscript**

Podstawowe typy danych oraz ich poprawne zastosowaniem / Złożone typy / Obiekty i interfejsy / Różnice między type oraz interface / Generyki / Typy warunkowe / Union types

## > **Wnioskowanie typów**

Mechanizm wnioskowania typów / Zawężanie typów / Kontrola przepływu i zaawansowane konstrukcje

# Agenda

Co nas czeka?

- > **Projektowanie typów**  
Tworzenie czytelnych i bezpiecznych typów /  
Modularyzacja typów i ponowne ich użycie / Interfejsy i  
relacje między typami
- > **Klasy i programowanie obiektowe w TS**  
Definiowanie klas i dziedziczenie / Abstrakcje i interfejsy /  
Modyfikatory dostępu
- > **Dobre praktyki**  
Bezpieczne obsługiwanie błędów / Konstrukcje try-catch  
/ Obsługa asynchroniczności / Testowanie w Typescript /  
Typowanie bibliotek zewnętrznych



# Nasza aplikacja

---

[github.com/matwjablonski/zus-typescript-2024-11](https://github.com/matwjablonski/zus-typescript-2024-11)



## Plan dnia

Co i kiedy?

- **8:00**  
zaczynamy
- **9:45 - 10:00**  
przerwa na kawę
- **11:45 - 12:15**  
przerwa obiadowa
- **14:00 - 14:15**  
przerwa na kawę
- **16:00**  
koniec

# **wprowadzenie do TypeScript**

Czym jest TypeScript? / Podstawy konfiguracji projektu / Omówienie komplikacji TypeScript do JavaScriptu i mapowanie kodu źródłowego /TypeScript vs JavaScript



# Czym jest TypeScript?

- język programowania
- nadzbiór (superset) dla języka Javascript
- dodaje statyczne i silne typowanie
- kompliuje się do Javascriptu
- można go używać z WebAssembly (AssemblyScript) oraz środowiskach takich jak Deno, ts-node
- wprowadzony na rynek przez Microsoft w roku 2012

# TypeScript w danych

TypeScript zaczął powstawać w 2010 roku. W 2024 roku blisko 40% projektów wykorzystuje TS zamiast JS.



## 2010

Anders Hejlsberg  
(Microsoft) rozpoczyna pracę na projektem



## 2012

Wydano TypeScript w wersji 0.8



## 2016

Angular 2.0 wydany w TypeScript

# Zalety TypeScriptu

- dodatkowa para oczu, która patrzy razem z nami na kod i szuka błędów
- świetne podpowiadanie składni oraz współpraca narzędzi na znacznie wyższym poziomie niż w czystym JS
- refaktoryzacja to nie problem (o spójność kodu dba kompilator)
- pewność zawartości stałych i zmiennych w programie
- typy w TS to dokumentacja kodu, która zawsze jest aktualna
- TS to sposób na zapisywania kontraktów / ustaleń z BE i innymi komponentami w aplikacji

# wady TypeScriptu

- sprawdza typy tylko w czasie komplikacji
- bywa niechlujny (w szczególności w przypadku bibliotek open-source)
- nie rozwiązuje problemów JSa
- dodatkowa złożoność dla programistów (krzywa uczenia się) - zmiana nawyków z JSa oraz konieczność przyswojenie nowych koncepcji
- wymaga doinstalowania kompilatora do środowiska programistycznego

# Dynamiczne typowanie

- nadawanie zmiennym typów w czasie działania programu
- zmienne nie posiadają typów przypisanych, otrzymują je dopiero w czasie kompilacji
- zmienna w różnych momentach wykonywania może przechowywać wartości różnych typów
- to wartość niesie typ a nie zmienna
- języki dynamicznie typowane to: Javascript, Python, Ruby

# Statyczne typowanie

- nadawanie zmiennym typów w czasie komilacji programu, poprzez ich deklarację
- raz nadany typ nie może zostać zmieniony
- pomaga uniknąć potencjalnych błędów związanych z typami (proste pomyłki typu zmiana string na number)
- ograniczenia nadane przez statyczne typowanie utrudniają zrobienie chaosu w kodzie

**"Jeśli chodzi jak kaczka i  
kwacze jak kaczka, to  
musi być kaczką."**

---



# Duck typing

- zakłada że typ obiekt sprawdzamy na podstawie zawartych w nim pól, a nie deklaracji typu
- wykorzystywane powszechnie w JS, ale spotykane również w TS
- misja Apollo13 - duck typing w praktyce

```
const a: unknown;
```

```
if (typeof a === "number" {  
}
```

# Konfiguracja

- ◆ **tsc**

kompilator języka Typescript

- ◆ **typescript**

pakiet zawierający język i kompilator języka  
Typescript

```
npm install typescript --save-dev  
npx tsc --init
```

# strict

Ustawiona na true włącza bardziej dokładne sprawdzanie poprawności typów. Włączenie tej opcji wiąże się z włączeniem wszystkich opcji, które należą do grupy strict.

## REGUŁA

strictBindCallApply

strictBuiltInIteratorReturn

strictFunctionTypes

strictNullChecks

strictPropertyInitialization

useUnknownInCatchVariables

## OPIS

TS sprawdzi czy funkcje wywoływane z **call**, **bind** i **apply** będą wywoływane z poprawnymi argumentami

wbudowane iteratory otrzymują typ **TReturn** zamiast **any** parametry funkcji są sprawdzane dokładniej

**null** i **undefined** otrzymują swoje własne, odrębne typy / błąd zostanie zwrocony, gdy będzie oczekiwana konkretna wartość

TS zwróci błąd, kiedy właściwość klasy zostanie zadeklarowana, ale nie zostanie jej ustawiona wartość w konstruktorze

błąd w **catch** w konstrukcji **try-catch** może przyjąć typ **unknown** zamiast **any**

# REGUŁA

noImplicitThis

alwaysStrict

noImplicitAny

# OPIS

**this** bez konkretnego typu jest niepoprawny zamiast przyjąć typ **any**  
wszystkie pliki są rozpatrywane w trybie **use strict** z ECMAScript  
wyrażenia bez określonego typu zamiast przyjąć typ **any**, zwrócią błąd

# holmplicitAny

W sytuacji, gdy typ zmiennej nie został określony i Typescript nie będzie mógł wywnioskować typu, powróci do typu bazowego **any**.

```
function fn(s) {  
    // Parameter 's' implicitly has an 'any' type.  
    console.log(s.substr(3));  
}
```

# Inne ustawienia

## REGUŁA

## OPIS

target

ustawia wersję JS do której będzie komplikowany kod TS

module

określa jaki system modułów został wybrany

exclude

określa tablicę nazw plików lub wzorców, które powinny być pomijane podczas rozwiązywania include

include

określa tablicę nazw plików lub wzorców do uwzględnienia w programie, nazwy plików są rozpoznawane względem katalogu zawierającego plik tsconfig.json

# Kompilacja do Javascript

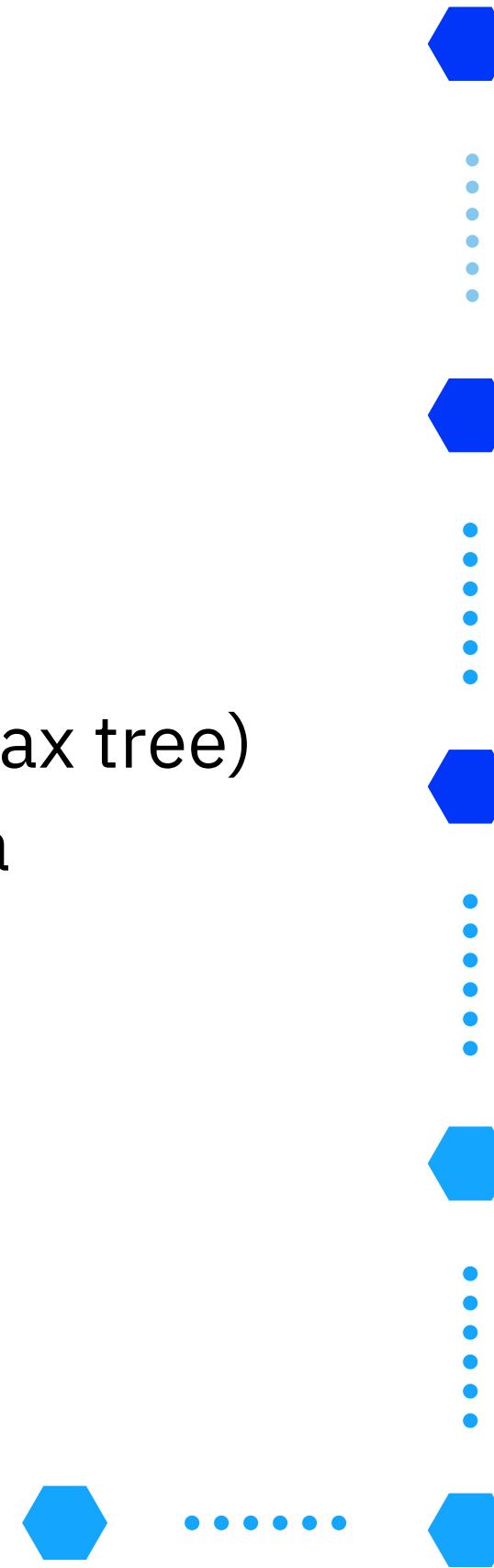
- sprawdzanie typów odbywa się przed komplikacją kodu do Javascript
  - system typów jest usuwany przekształcania Typescript AST (abstract syntax tree) do kodu Javascript
  - dodane typy nie mają wpływu na wygenerowany kod Javascript
  - ponieważ proces sprawdzania typów odbywa się tak wcześnie i typy nie są brane pod uwagę na kolejnym etapach komplikacji - **prawidłowy kod Javascript jest prawidłowym kodem**
- TypeScript**

# Proces

Od kodu TS poprzez AST (abstract syntax tree) do kodu JS i ostatecznie do środowiska uruchomienia owego

## Krok 6

Bytecode (kod bajkowy) jest przetwarzany przez środowisko uruchomieniowy



# C# i Java

- wykorzystują rzeczywisty, nominalny system typów
- każda wartość lub obiekt ma tylko jeden dokładny typ
- definicja typu znajduje się w klasie
- danego typu nie możemy używać poza klasą (chyba, że zachodzi proces dziedziczenia lub implementacji publicznego interfejsu)
- typy są dostępne również na etapie wykonywania kodu
- typy są powiązane poprzez deklaracje a nie poprzez wartości

# ...typy w Typescript?

- Typescript wykorzystuje typowanie strukturalne
- lepiej myśleć o typie w TS jako o zestawie wartości, a wartość o danym typie musi po prostu pokryć część struktury
- jedna wartość nie musi być przypisana do jednego typu, jednocześnie może spełniać wiele typów
- w TS obiekty nie są jednego typu, **obiekt spełniający dany interfejs może być użyty nawet tam, gdzie nie było deklarowanej relacji między nim a interfejsem, który spełnia**

# Mapowanie kodu źródłowego

- Typescript umożliwia tworzenie plików typu sourceMap
- podczas komplikacji do JS powstają wówczas pliki .js.map lub .jsx.map
- pliki map ułatwiają debuggowanie kodu TS poprzez powiązanie plików .ts z plikami .js

# ts-node

- narzędzie, które łączy kompilator Typescript oraz nodejs
- pozwala na bezpośrednie uruchomienie kodu napisanego w Typescript bez konieczności ręcznej komplikacji do JSa
- zastępuje użycie **tsc** i **node** jednym **ts-node**

# **zadanie**

15 minut

**1.**

zainicjuj projekt TS, skonfiguruj tsconfig.json tak aby TS kompilował wszystkie pliki z katalogu /src do katalogu /dist

# **System typów w Typescript**

Podstawowe typy danych oraz ich poprawne zastosowaniem / Złożone typy / Obiekty i interfejsy / Różnice między type oraz interface / Generyki / Typy warunkowe / Union types



**"...żaden system typów  
nie może całkowicie  
wyeliminować  
występowania bugów,  
może tylko wykazać ich  
obecność..."**



# Podstawowe typy

- boolean
- number
- string
- symbol
- bigint
- null
- undefined
- object
- Array

# a ponad to...

- tuple
- enum
- void
- any
- Object
- never
- unknown

# boolean

- przyjmuje wartość logiczną **true** lub **false**

# number

- przyjmuje wartość liczbową (liczby zmiennoprzecinkowe)
- możliwe jest używanie literałów heksadecymalnych, oktalnych i binarnych

```
const a: number = 123 + 0x0e2f + 0b1100 + 0o710;
```

```
// 123 + 3631 + 12 + 456 = 4222
```

# string

- ciągi znaków podawane w pojedynczym cudzysłowie ('), podwójnym ("") oraz template string (`)
- pojedynczy i podwójny mają to samo zastosowanie
- template string działa tak samo jak w dokumentacji JS - obsługa wielu linii oraz przekazywanie do nich wywołań funkcji i zmiennych

# symbol

- gwarantuje unikalność
- zostały wprowadzone do ES2015
- symbole nie są enumerowalne

# bigint

- przeznaczony do reprezentowania dowolnie dużych liczb całkowitych
- mieszanie typów number i bigint w JS kończy się rzuceniem wyjątku, a w TS błędem kompilacji

# null i undefined

- typy przydatne do opisywania wartości opcjonalnych albo “nullowalnych”

# Array

- typ opisuje tablice
- jest typem złożonym - wszystkie typy złożone są porównywane przez referencję a nie wartość co oznacza, że `[] === []` zwróci **false**
- Array jest typem generycznym **Array<T>**
- istnieją dwa sposoby zapisu: **Array<T>** oraz **T[]**

# tuple

- tupla to skończona lista elementów, w TS jest to tablica o określonej na sztywnie liczbie elementów o określonych typach na konkretnych pozycjach

```
const interval1: [number, string] = [0, "hour"]; // OK  
const interval2: [number, string] = []; // Błąd
```

# enum

- enum to zbiór nazwanych wartości
- enumy w TS są domyślnie liczbami rozpoczynającymi się od 0
- możliwe jest tworzenie enumów o wartościach będącymi stringami

```
enum UserRole {  
    Admin = "admin",  
    User = "user",  
}  
const role: UserRole = UserRole.Admin; // admin
```

# void

- oznacza brak wartości
- używany do oznaczania funkcji, które nic nie zwracają
- do zmiennej o typie void można przypisać tylko wartość undefined

```
function fn1(): void {}  
function fn2(): void { return; }  
function fn3(): void { return undefined; }
```

# any

- domyślny typ
- oznacza dowolną wartość
- użycie **any** to zaprzeczenie statycznego i silnego typowania - taki wentyl bezpieczeństwa
- jeśli użyjemy **any** do opisania obiektu to wszystkie jego pola i metody również będą domyślnie typu **any**
- **any** oznacza że typ Cię nie obchodzi, a nie że go nie znasz

# never

- typ opisujący wartość, która nigdy nie wystąpi
- przydatny do funkcji, które nigdy nic nie zwracają - tylko rzucają wyjątek lub które się zawieszają (infinite loop)
- pozwala też wyłapać sytuacje, które nie powinny się zdarzyć

# unknown

- typ powstał by przestano nadużywać typu any
- reprezentuje typ, który jest nieznany
- do unknown mogę przypisać dowolną wartość, ale nie mogą nic z niego odczytać
- nie możemy przypisać zmiennej o typie unknown do innej zmiennej z określonym jawnie typem

# object i Object

- typ **object** reprezentuje wszystko co nie jest typem prymitywnym (string, number, bigint, symbol, null i undefined)
- typ **Object** opisuje właściwości i metody, które są wspólne dla wszystkich obiektów w JS
- typ **Object** jest używany głównie do dziedziczenia

# type

- w TS możemy definiować własne typy
- przyjęto się że nazwy typów piszemy wielką literą
- własne typy są idealne do opisywania kształtu oczekiwanych obiektów
- pozwalają na tworzenie aliasów typów

```
type User = {  
    name: string;  
    age: number;  
}  
  
const user: User = { name: "Mateusz", age: 35 };
```

# **zadanie**

15 minut

**1.**

przygotuj typ Book, który będzie opisywał takie cechy tej książki jak tytuł, autor, data wydania, recenzje oraz kategorię

**2.**

przygotuj enum dla kategorii: dostępne to romans, kryminał, reportaż i biografia

**3.**

połącz dane z pliku data.ts ze swoimi typami

# Argumenty funkcji

- w TS typy argumentów funkcji są obowiązkowe i muszą być podane
- funkcje deklarowane są w taki sam sposób jak w JS

```
function multiply(x: number, y: number) {  
    return x * y;  
}
```

# Typ zwracany

- każda funkcja ma także typ zwracany
- TS potrafi wnioskować typ na postawie operacji, która znajduje się po słowie return
- wnioskowanie typu to inferencja
- jeśli funkcja nie zwraca niczego, używamy typu void

```
function multiply(x: number, y: number): number {  
    return x * y;  
}
```

```
function n(): void {}
```

# Typ funkcji

- nazwy parametrów w typie i w implementacji mogą być różne - liczy się tylko typ
- do napisania typu funkcji użyjemy słowa kluczowego **type**
- do parametrów funkcji oznaczonej typem nie musimy dopisywać typów, TS dokona inferencji (wnioskowania) na podstawie typu funkcji

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

```
const multiply: MultiplyFunction = (a, b) => a * b;
```

# Parametry opcjonalne

- w JS wszystkie parametry funkcji traktowane są jak opcjonalne, nie podanie ich nie powoduje błędu
- w TS każdy argument jest wymagany, ale może zostać oznaczony jako opcjonalny
- opcjonalne argumenty muszą znaleźć się na końcu listy argumentów
- oznaczając argument o typie **string** jako opcjonalny otrzyma on typ: **string | undefined**

```
function getFullName (  
    firstName: string,  
    middleName?: string,  
    lastName?: string;  
) => // jakaś logika
```

# Parametry domyślne

- zarówno w JS, jak i w TS parametry mogą przyjmować wartości domyślne
- na podstawie domyślnych wartości TS będzie wnioskował jaki typ powinien mieć argument
- podanie domyślnej wartości powoduje, że argumenty stają się opcjonalne

```
function multiply (x = 2, y = 1) {  
    return x * y;  
}
```

```
multiply();
```

# Funkcje wariadyczne

- to taki rodzaj funkcji, który potrafi przyjmować nieokreśloną liczbę parametrów
- do określania takiej parametrów funkcji służy **spread operator** (...) w notacji zwanej **rest parameters**
- typ **rest parameters** to tablica wartości o jakimś generycznym typie

```
function sum (...numbers: number[]) {  
    return numbers.reduce((a, b) => a + b, 0);  
}
```

```
sum(1, 2, 3, 4, 100); // 110
```

# **zadanie**

15 minut

**1.**

wykonaj zadania od task-1 do task-6

# Przeciążenie funkcji

- przeciążenie funkcji (function overloading) to cecha programowania pozwalająca posiadać więcej niż jedną funkcję o tej samej nazwie, ale z różnymi wariantami przekazywanych argumentów
- JS nie oferuje sposobu przeciążania funkcji znanego z innych języków, takich jak C#
- w JS istnieje kilka obejść na wykonanie przeciążenia funkcji, ale żadne nie jest oficjalnym i intuicyjnym podejściem
- w TS przeciążenie funkcji istnieje - parametry mogą się różnić ilością i/lub typami
- typ podany w implementacji musi być zgodny ze wszystkimi przeładowaniami (przeciążeniami), ale sam w sobie nie tworzy nowego przeciążenia

# Przeciążenie funkcji

```
declare function getWidget(n: number): Widget;  
declare function getWidget(s: string): Widget[];
```

```
let x: Widget = getWidget(43);  
let arr: Widget[] = getWidget("all of them");
```

# Przeciążenie funkcji

```
function getWidget(arg: number | string): Widget | Widget[] {  
    if (typeof arg === 'number') {  
        return "Widget" as Widget;  
    } else {  
        return ["Widget1", "Widget2"] as Widget[];  
    }  
}
```

# this

- TS traktuje **this** jako specjalny argument przekazywany do funkcji
- w celu opisania jego typu należy umieścić go na pierwszym miejscu w deklaracji przed innymi parametrami
- jeśli nie chcemy używać **this** wówczas możemy określić jego typ jako **never**

```
function fn(this: { name: string }, arg: string) {}  
const object = {  
    name: 'test',  
    fn,  
}  
object.fn('Hello');
```

# Interface

- podlega zasadom typowania strukturalnego
- nazywa i definiuje typ bez implementacji
- historycznie w TS był to jedyny sposób deklarowania typów
- interfejsy można rozszerzać (**extends**) oraz implementować (**implements**) w klasach
- interfejsy podlegają mechanizmowi łączenia deklaracji (**declaration merging**)
- interfejsy nie pozwalają na tworzenie aliasów (w przeciwieństwie do **type**)

# Interface

```
interface Person extends Human {}
```

```
class PersonClass implements Person {}
```

# declaration merging

- dwa interfejsy o tej samej nazwie zostaną połączone i razem będą tworzyć dany typ
- mechanizm ten przydaje się, gdy deklaracje pochodzą z różnych źródeł, w szczególności gdy istnieje możliwość, że w naszym kodzie będziemy rozszerzać typ zewnętrznej biblioteki

```
interface A {  
    a: string;  
}  
  
interface A {  
    b: number;  
}
```

# readonly

- służy do oznaczania właściwości w klasach, typach i interfejsach jako tylko do odczytu
- właściwość oznaczona jako readonly jest dostępna poza klasą do odczytu, ale nie można jej modyfikować
- oznaczenie readonly nie jest przenoszone do kodu JS

```
interface Employee {  
    readonly employeeCode: number;  
    name: string;  
}
```

# as const

- w JS wszystko jest domyślnie mutowalne
- deklaracja zmiennej w JS za pomocą **const** powoduje, że referencja staje się niemutowalna
- deklaracja **as const** w TS powoduje, że obiekty i tablice nie mogą być rozszerzane
- tablice oznaczone jako **as const** zamieniane są na tuple

```
let arr = [1, 2, 3] as const;  
arr.push(102); // error
```

```
let obj = { a: 1 } as const;  
obj.b = 3; // error
```

# **zadanie**

15 minut

**1.**

wykonaj zadania od task-7 do task-10

# Typy generyczne

- określane też generykami, typami polimorficznymi lub szablonami
- typ generyczny zakłada, że jakąś część typu może być sparametryzowana
- zapis **Typ<InnyTyp>** oznacza typ generyczny

```
const list: Array<string> = ["a", "b", "c"];
```

```
type User = { name: string };
const userList: User[] = [{ name: "Olek" }];
```

```
type A<T> = { value: T, name: string }
```

# Funkcje generyczne

```
const id = <T>(x: T): T => x;  
  
const result = id<number>(1)  
  
function value<R>(v: R): R {  
    return v;  
}
```

# Generyczne klasy

```
class Person<T> {  
    equipment: Array<T> = [];  
  
    push(item: T) {  
        equipment.push(item);  
    }  
}
```

# Generyczne interfejsy

```
interface Person<T> {  
    equipment: T[];  
    name: string;  
}
```

# Unia (union type)

- unia to taki typ, który zawiera wartości wspólne dla typów składających się na nią
- unie oznaczamy za pomocą operatora |
- unia z **undefined** oznacza wartość opcjonalną, z **null** nullowalną

```
type A = { a: string; b: number; }
type B = { b: number; c: string; }
type U = A | B;
declare const union: U;
union.a; // błąd
union.b; // ok
```

# Intersection type

- intersection to taki typ, który zawiera wszystkie wartości z typów składających się na niego
- intersection oznaczamy za pomocą operatora **&**
- przydatny do opisywania takich operacji jak **Object.assign** czy **extends**

```
type A = { a: string; b: number; }
type B = { b: number; c: string; }
type Inter = A & B;
declare const intersection: Inter;
intersection.a; // ok
intersection.b; // ok
```

Zarówno union type, jak i  
intersection type  
powinniśmy rozpatrywać  
jako zbiory typów.



# **zadanie**

15 minut

**1.**

wykonaj zadania od task-11 do task-13

**2.**

do wykonania zadania 14 wykorzystaj metody  
do zarządzania elementami DOM,  
createElement, appendChild

**3.**

pamiętaj o typowaniu struktur DOM

# Typy warunkowe

- typy warunkowe pozwalają nad dobrać typ w zależności od sprawdzanego warunku
- wszystkie strony operacji zwracają typy i są typami
- do sprawdzania warunków służy słowo kluczowe **extends**

```
type IsBoolean<T> = T extends boolean ? true : false;
```

```
type Check = IsBoolean<boolean>; // true
type Check2 = IsBoolean<string>; // false
```

# Typy warunkowe na unii

- typy warunkowe możemy wykonywać również na unii

```
type NonNullable<T> = T extends null | undefined ? never : T;
```

```
type TOne = NonNullable<number>; // number
```

```
type TTTwo = NonNullable<null | undefined>; // never
```

# **wnioskowanie typów**

Mechanizm wnioskowania typów / Zawężanie typów / Kontrola przepływu i zaawansowane konstrukcje



# Inferencja

- inferencja typów to mechanizm wnioskowania typów
- TS wnioskuje typy na podstawie przypisania wartości do zmiennej, na podstawia ciała funkcji (zwracanych wartości, wykonywanych operacji)
- należy pamiętać, że inferencja czasami zawodzi i to do nas należy dookreślenie typu
- z inferencji należy korzystać uważnie, dobrze użyta pozwoli nam uniknąć wpisywania typów wszędzie i jednocześnie zagwarantuje wysoki poziom bezpieczeństwa

# Inferencja dla const i let

```
let b = 123; // number  
const a = 123; // '123'
```

# Narrowing

- **narrowing** typów to mechanizm zawężania typów
- zawężanie odnosi się do procesu zmniejszania zakresu typu z szerszego typu do bardziej specyficznego typu w określonym bloku kodu lub kontekście
- jeżeli nie ustawimy odpowiednich zabezpieczeń TS będzie nas ostrzegał, gdy istnieje ryzyko, że typ jest zbyt szeroki
- do **narrowing'**u są wykorzystywane **type guard'**y, np: **typeof** czy **instanceof**

# Kontrola przepływu

- analiza przepływu sterowania w TypeScript odnosi się do procesu, w którym kompilator TypeScript analizuje przepływ kodu i wyciąga wnioski na temat typów zmiennych w różnych punktach programu, najczęściej wykorzystując do tego konstrukcje if / else if
- umożliwia TypeScript'owi zawężenie typów zmiennych w określonych blokach kodu, w oparciu o warunki i instrukcje przepływu sterowania
- pomaga to uczynić system typów bardziej ekspresyjnym i wychwycić potencjalne błędy typów

# Type guards

- type guard to blok kodu sprawdzający jaki jest typ przekazanej zmiennej
- istnieją zdefiniowane guardy, tj: typeof, instanceof, in
- oprócz powyższych do tworzenia type guardów można użyć porównań lub tworzyć własne funkcje sprawdzające

# typeof

- jest wbudowanym operatorem Javascript
- nie porównujemy obiektu do instancji danej klasy, ale porównujemy typ (jego nazwę określoną w JS)

# instanceof

- wbudowany w Javascript operator
- sprawdza czy nasza zmienna jest instancją wybranej przez nas klasy
- metoda ta opiera się na porównaniu łańcuchów prototypów (prototype chain) dla zmiennej oraz klasy



- dostarcza informacji czy obiekt, który sprawdzamy, ma w sobie określone właściwości
- właściwość znajdująca się w naszym warunku, powinna być unikalna dla jednego konkretnego typu, aby zawęzić wyboru TSA do jednej opcji

# **zadanie**

15 minut

**1.**

wykonaj zadania od task-14 do task-17

# **Klasy i programowanie obiektowe**

Definiowanie klas i dziedziczenie / Abstrakcje i interfejsy / Modyfikatory dostępu



# Klasy

```
class Car {  
    private name: string;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
    public getName() {  
        return this.name;  
    }  
}
```

```
const car = new Car('Audi');  
car.name; // Error: Property 'name' is  
private and only accessible within class  
'Car'  
car.getName(); // Audi
```

# Modyfikatory dostępu

- ◆ **public**

- ◆ **protected**

- ◆ **private**

```
class Car {  
    protected name: string;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
  
    public getName() {  
        return this.name;  
    }  
}
```

```
class ECar extends Car {  
    constructor(name: string) {  
        super(name);  
    }  
  
    const eCar = new ECar('VW');  
    eCar.getName(); // VW  
    car.name; // Error: // Property 'name' is  
    protected and only accessible within  
    class 'Car' and its subclasses.
```

```
class Car {  
    protected name: string;  
    readonly minEcoRate = 0.8;  
  
    constructor(name: string) {  
        this.name = name;  
        this.minEcoRate = 10; // Type '10' is not assignable to type '0.8'  
    }  
    public getName() {  
        return this.name;  
    }  
}
```

```
class Car {  
    protected name: string;  
    readonly minEcoRate = 0.8;  
    private _hasPassengers;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
    public getName() { return this.name; }  
    get hasPassengers(): boolean { return this._hasPassengers; }  
    set hasPassengers(value: boolean) { this._hasPassengers = value; }  
}
```

```
interface Thing {  
    setName(): void;  
    getName(): string;  
}  
  
class Box implements Thing {  
    private name: string;  
  
    public setName(): void {}  
  
    public getName(): string { return this.name; }  
}
```

# **zadanie**

20 minut

**1.**

wykonaj zadania od task-18 do task-22

# Klasy abstrakcyjne

- mogą zawierać metody abstrakcyjne, które nie mają implementacji i które muszą być zaimplementowane przez podklasy
- mogą zawierać zwykłe metody (już w implementacji)
- metody abstrakcyjne zapewniają, że każda podklasa zapewnia własne specyficzne zachowanie dla metody

```
abstract class Animal {  
    abstract makeSound(): void; // Abstract method, no implementation  
  
    move(): void { console.log("Moving..."); }  
}
```

# Pole prywatne ES

- w JS pola prywatne opisujemy za pomocą #
- pole prywatne w TS nie jest tym samym czym pole prywatne w JS
- przed wprowadzeniem # pola prywatne oznaczano \_, ale nie były to pola prywatne znane z innych języków obiektowych

# Pola i metody statyczne

- klasy pozwalają na tworzenie pól i metod statycznych, które mogą być wykonywane na klasach a nie na instancjach
- pola i metody statyczne poprzedzamy słowem kluczowym **static**
- deklaracja klasy tworzy dwa typy, np dla klasy **User**:
  - **typeof User** - zawiera wszystkie statyczne metody i pola klasy
  - **User** - oznacza instancję klasy

# Type casting

- stosujemy, gdy wiemy lepiej niż TS jaki typ powinien być przypisany do danej zmiennej
- najczęściej wiedza taka wynika z logiki biznesowej a nie samego kodu
- castowanie możemy wykonać używając słowa kluczowego **as (variable as string)** lub z wykorzystaniem **<>**, przykładowo **<string>variable**

# Force type casting

- stosujemy, gdy nadpisanie za pomocą **as** nie jest możliwe, ponieważ typy są od siebie zbyt różne (TS zrzuca nam błąd)
- aby wymusić zmianę typu pomimo błędu zmieniamy typ najpierw na **unknown** a następnie na typ oczekiwany, np. **variable as unknown as string**

# **zadanie**

20 minut

**1.**

wykonaj zadania od task-23 do task-26

# Type predicate

- występuje w postaci **<zmienna> is <typ>**
- jeśli funkcja zwróci wartość **true** wówczas TS wie jakiego typu jest zmienna
- doskonałe narzędzie do zawężania typów i tworzenia własnych type guardów
- zaleca się ostrożne korzystanie z type predicate ponieważ łatwo popełnić błędy podczas jego tworzenia

# Infer

- operator **infer** w TypeScript jest używany w typach warunkowych, aby wywnioskować typ zmiennej na podstawie struktury innego typu
- dzięki słowu kluczowemu **infer** możemy w pewnym sensie ręcznie sterować inferencją typów

# Unie dyskryminacyjne

- to połączenie takich typów, z których każdy posiada jedno wspólne pole, na podstawie którego możemy określić, z jakim z nich mamy do czynienia
- przydatne gdy trudno

# **zadanie**

20 minut

**1.**

wykonaj zadania od task-27 do task-29

# Projektowanie typów

Tworzenie czytelnych i bezpiecznych typów / Modularyzacja typów i ponowne ich  
użycie / Interfejsy i relacje między typami



# Enumy

- podczas tworzenia enumeracji nie musimy podawać wszystkich nowych wartości liczbowych - TS kolejne elementy zwiększy o jeden
- enumy są obiektami i tak powinniśmy je traktować
- koncepcyjnie obiekt, który powstaje po komplikacji zawiera właściwości zarówno jako kolejne liczby oraz przypisane im nazwy (taki podwójny zapis ułatwia konwersji w dwie strony)
- wartością enuma może być również wynik wykonania wyrażenia, ale wówczas kolejne wartości też muszą być opisane
- typy enumów są sprawdzane nominalnie, oznacza to, że nie możemy przypisać wartości z jednego enuma do zmiennej typu innego enuma - nawet gdy mają taką samą strukturę
- enumeracje są kompatybilne z obiektami a zatem tak gdzie są wymagane obiekty możemy używać również enumeracji

# const enum

- zwykły enum jest komplikowany do JS w postaci “małego potworka” obiektu, a miejsca w których był użyty mają odwołanie do niego
- istnieje możliwość zapisu const enum, który nie pozostawia po sobie śladu w JS (pozostają tylko wartości i są opatrzone komentarzem)

# const enum

```
const enum Role {  
    A,  
    B,  
    C,  
}  
const allowedRoles = [Role.A, Role.B]  
  
// po komplikacji  
const allowedRoles = [0 /* A */, 1 /* B */]
```

# Test wyczerpania

- mechanizm w TS, który będzie wymagał od nas pokrycia wszystkich, które należy uwzględnić podczas rozpatrywania typu (np. wykonując switch() na enumeracji)
- test wyczerpania działa tylko z włączonym **noImplicitReturns**

# Mapped types

- ten mechanizm w TS pozwala na budowanie typów bazując na innych typach (bez konieczności powtarzania się wielokrotnie)
- Mapped types bazują na składni sygnatur indeksów, których używają do deklarowania typów właściwości, które nie były wcześniej zadeklarowane
- innymi słowy: nie musimy opisywać wszystkich pól w typie, gdy są do siebie podobne
- służą nam do tego operatory **keyof** oraz **in keyof**

# keyof

- służy do pobierania pól z typu
- zwraca unię literałów

```
type Player = {  
    name: string;  
    xp: number;  
}
```

```
type PlayerKeys = keyof Player; // 'name' | 'xp'
```

# in keyof

- połączenie operatora keyof oraz operatora in z sygnaturą indeksu pozwala na stworzenie nowego typu, który będzie kopią już istniejącego
- w poniższym przykładzie K to kolejne pola z typu Player
- takie zapis tworzy homomorfizm (gr. podobny kształt)

```
type Player = {  
    name: string; xp: number;  
}  
type Player2 = {  
    [K in keyof Player]: Player[K]  
}
```

+ i -

- TS wprowadza dodanie dodatkowych operatorów do typów, służących do usuwania lub dodawania takich cech jak np. **readonly** lub **?**
- operator **+** dodaje cechę, natomiast **-** odejmuje

```
type Player = {  
    readonly name: string; xp?: number;  
}  
type Player2 = {  
    -readonly [K in keyof Player] -?: Player[K]  
}
```

# Mapowane generyki

- typów mapowanych możemy używać również z typami generycznymi

```
type Partial<T> = {  
  [P in keyof T]?: T[P]  
}  
  
type Required<T> = {  
  [P in keyof T]-?: T[P]  
}
```

# **zadanie**

20 minut

**1.**

wykonaj zadania od task-30 do task-32

# Typy nominalne

- TS opiera się o typy strukturalne a zatem sprawdzany jest tylko kształt typu a nie jego przeznaczenie, co oznacza, że łatwo o proste błędy - np. użycie sekund tam gdzie są potrzebne milisekundy
- w społeczności pojawiły się głosy, aby wprowadzić możliwość typowania nominalnego, ale nadal nie ustalono wspólnego frontu jak to zrobić i na dziś typów nominalnych nie ma

# Branding

- idea “brandowania” polega na dodaniu do typu dodatkowego unikalnego pola, które pozwoli na odróżnienie go od innych podobnych pól, z założenie, że nigdy nie przypiszemy do niego żadnego wartości, a jego deklaracja to tylko wskazówka dla kompilatora
- pola brandowane są prefixowane \_\_

# Branding

```
type InvoiceID = number & { __brand: "InvoiceID" };
```

```
type UserID = number & { __brand: "UserID" };
```

```
declare let invoiceId: InvoiceID;
```

```
declare let userId: UserID;
```

```
invoiceId = userId; // Błąd
```

# Brand<T, BrandT>

```
type Brand<T, BrandT> = T & { __brand: BrandT };
```

```
type InvoiceID = Brand<number, "InvoiceID">;
```

# Flavoring

- idea podobna do brandingu, ale pozwalająca używać również typów strukturalnych
- polega na dodaniu specjalnego pola **\_\_flavor**, ale z założeniem że będzie ono opcjonalne

# Flavor<T, FlavorT>

```
type Flavor<T, FlavorT> = T & { _flavor?: FlavorT };
```

```
type InvoiceID = Flavor<number>;
```

# Brand i Flavor

- branding zalecany jest do stosowania w typach prostych, a flavoring natomiast w klasach i obiektach - taki podział zaspokaja większość potrzeb
- poprzez opcjonalność (flavor) pozwala na dopisanie dowolnych wartości do zmiennej - nie zawsze będzie to pożądane, np. przy określaniu ID

# **zadanie**

20 minut

**1.**

wykonaj zadania od task-33 do task-34

# Utility types

- zestaw narzędzi dostarczonych przez Typescript do obsługi najczęściej występujących transformacji na typach

# Awaited<Type>

- ten typ ma na celu modelowanie operacji takich jak **await** w funkcjach asynchronicznych lub metody **.then()** w obietnicach
- pomaga wyciągnąć typ, który zwraca **Promise**

```
async function fetchData(): Promise<string> {  
  return "Hello, world!";  
}
```

```
type ResultType = Awaited<ReturnType<typeof fetchData>>;  
const data: ResultType = "Hello, world";
```

# Partial<Type>

- sprawia, że wszystkie pola w danym typie stają się opcjonalne

```
interface Todo {  
    title: string;  
    description: string;  
}  
  
function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {  
    return { ...todo, ...fieldsToUpdate };  
}
```

# Required<Type>

- sprawia, że wszystkie pola w danym typie stają się wymagane

```
interface Props {  
  a?: number;  
  b?: string;  
}  
const obj: Props = { a: 5 };  
const obj2: Required<Props> = { a: 5 }; // błąd
```

# Readonly<Type>

- ustawia wszystkie pola jako readonly
- wartości nie mogą być nadpisane / zmienione
- przydatne dla opisywania wyrażeń, które w runtime będą niemutowalne (**Object.freeze**)

```
interface Todo {  
    title: string;  
}  
  
const todo: Readonly<Todo> = {  
    title: "Delete inactive users",  
};  
  
todo.title = "Hello"; // błąd
```

# Record<Keys, Type>

- narzędzie do mapowania typów
- **Keys** to właściwości w obiekcie
- **Type** to typ jakie te właściwości mogą przyjąć

```
type Result = Record<'car' | 'bike' | 'bus', boolean>
```

```
const obj: Result = {  
    car: true,  
    bike: false,  
    bus: false,  
}
```

# Pick<Type, Keys>

- narzędzie do wybierania właściwości (**Keys**) z typu w celu utworzenia zawężonego typu

```
interface Todo {  
    title: string;  
    description: string;  
    completed: boolean;  
}  
type TodoPreview = Pick<Todo, "title" | "completed">;
```

# Omit<Type, Keys>

- narzędzie do pomijania właściwości (**Keys**) z typu w celu utworzenia zawężonego typu

```
interface Todo {  
    title: string;  
    description: string;  
    completed: boolean;  
    createdAt: number;  
}  
type TodoPreview = Omit<Todo, "description">;
```

# **Exclude<UnionType, ExcludedMembers**

- narzędzie do wykluczania elementów z uni w celu zawężenia typu

```
type T2 = Exclude<string | number | (() => void), Function>;
```

```
// T2 = string | number
```

# Extract<Type, Union>

- narzędzie do wyciągania elementów z uni w celu zawężenia typu

```
type T2 = Extract<string | number | (() => void), Function>;
```

```
// T2 = () => void
```

# NonNullable<Type>

- narzędzie do usuwania null lub undefined z typu

```
type T0 = NonNullable<string | number | undefined>;  
// type T0 = string | number
```

```
type T1 = NonNullable<string[] | null | undefined>;  
// type T1 = string[]
```

# Parameters<Type>

- narzędzie zwraca parametry z funkcji w postaci tupli

```
type T1 = Parameters<(s: string) => void>;  
// type T1 = [s: string]
```

```
type T2 = Parameters<<T>(arg: T) => T>;  
// type T2 = [arg: unknown]
```

# ConstructorParameters <Type>

- narzędzie zwraca parametry z konstruktora z klasy w postaci tupli lub tablicy

```
class C {  
    constructor(a: number, b: string) {}  
}  
  
type T3 = ConstructorParameters<typeof C>;  
  
// type T3 = [a: number, b: string]
```

# ReturnType<Type>

- narzędzie zwraca typ zwracany z typu funkcji do niego przekazanej

```
type T1 = ReturnType<(s: string) => void>;
```

```
// T1 = void
```

# InstanceType<Type>

- narzędzie zwraca typ instancji z typu konstruktora funkcji / klasy do niego przekazanej

```
class C {  
    x = 0;  
    y = 0;  
}
```

```
type T0 = InstanceType<typeof C>;  
// T0 = C
```

# NoInfer<Type>

- blokuje inferowanie do wskazanego typu
- **NoInfer<Type>** oraz **Type** są identyczne

```
function createStreetLight<C extends string>(  
    colors: C[],  
    defaultColor?: NoInfer<C>,  
) {}  
  
createStreetLight(["red", "yellow", "green"], "red"); // OK  
createStreetLight(["red", "yellow", "green"], "blue"); // Error
```

# ThisParameterType<T>

- zwraca typ **this** jeśli this jest otypowany w funkcji lub **unknown** jeśli nie został określony

```
function toHex(this: Number) {  
    return this.toString(16);  
}
```

```
function numberToString(n: ThisParameterType<typeof toHex>) {  
    return toHex.apply(n);  
}
```

# **zadanie**

20 minut

**1.**

wykonaj zadania od task-35 do task-37

# Dobre praktyki

Bezpieczne obsługiwanie błędów / Konstrukcje try-catch / Obsługa asynchroniczności / Testowanie w Typescript / Typowanie bibliotek zewnętrznych



# try...catch

```
try { /* ... */ }
catch (e: unknown) {
  e.message // errors
  if (typeof e === "string") {
    e.toUpperCase() // works, `e` narrowed to string
  } else if (e instanceof Error) {
    e.message // works, `e` narrowed to Error
  }
  // ... handle other error types
}
```

# async

```
async function fetchData<T>(url: string): Promise<T> {  
    const response = await fetch(url);  
  
    if (!response.ok) {  
        throw new Error(`Failed to fetch data: ${response.statusText}`);  
    }  
  
    const data: T = await response.json();  
    return data;  
}
```

# .d.ts

- są to pliki definicji Typescripta, które pozwalają dopisać typy do rozwiązań istniejących w JS
- pakiety typów w NPM są umieszczane w rejestrze w grupie **@types**
- w przypadku dodania deklaracji która będzie dotyczyła istniejącego modułu - TS je ze sobą połączy

```
declare module "*svg" {  
    const content: FC<React.SVGAttributes<SVGElement>>;  
    export default content;  
}
```

# Poprawne typowanie Array#filter

```
type Item = { value?: number }
declare const items: Array<Item>;
items
  .filter((item): item is { value: number } => typeof item.value !== 'undefined')
  .map(item => {
    // tutaj nadal item.value nie jest już opcjonalne
  })
```

# Polecane narzędzia

- zod - silnie typowana biblioteka do walidowania formularzy
- ts-node - kompilator TS razem ze środowiskiem uruchomieniowym node
- ts-essentials - zestaw przygotowanych przydatnych typów do użycia w aplikacjach
- DefinitelyTyped - repozytorium zawierające typy dla najpopularniejszych bibliotek JS

# Testowanie typów

- podejścia są różne / testować i nie testować
- pomaga nam uniknąć błędów związanych z nieprawidłowymi typami danych
- testowanie typów zwiększa czytelność i zrozumiałość kodu, ponieważ precyzyjnie określamy oczekiwane typy dla zmiennych i funkcji

# Testowanie

- Jest - unit testing
- Cypress / Playwright - e2e oraz component testing
- tsd – narzędzie do generowania deklaracji typów TypeScript na podstawie istniejącego kodu JavaScript. Dzięki temu możemy łatwo rozpoczęć testowanie typów w istniejących projektach.
- ts-jest – Ts-jest to integracja TypeScript z biblioteką testową Jest. Pozwala ona na testowanie typów wraz z innymi testami jednostkowym

# Czy Typescript to nasza przyszłość?

---



# Alternatywy?

- JSDoc
- Flow
- LiveScript, bazuje na CoffeeScript
- Fable, czyli F# do JavaScript
- PureScript, inspirowany Haskelllem
- Kotlin/JS

# KAHOOT

20 minut



# ANKIETA

[sages.link/290284](https://sages.link/290284)





# Dziękuję za uwagę

---

[mateusz.jablonski@sages.io](mailto:mateusz.jablonski@sages.io)  
[mail@mateuszjablonski.com](mailto:mail@mateuszjablonski.com)

[mateuszjablonski.com](http://mateuszjablonski.com)