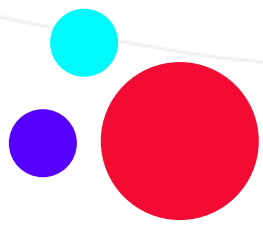


# Typescript

TypeScript – wprowadzenie do typowania kodu



# HELLO

## Dariusz Sibik

Senior Frontend Software Engineer (Spyrosoft)





TypeScript jest nadzbiorem Javascript i rozszerza o możliwość typowania.

TypeScript oferuje wszystkie funkcje JavaScript, a oprócz tego dodatkową warstwę typów.

Twój istniejący działający kod JavaScript jest również kodem TypeScript.

Kod napisany w Typescript jest transpilowany do Javascript.



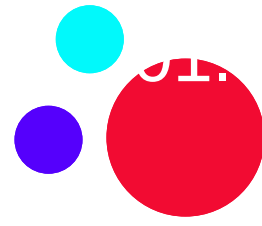
# Zalety Typescript

Główną zaletą języka TypeScript jest to, że może wyróżnić nieoczekiwane zachowanie w kodzie, zmniejszając prawdopodobieństwo wystąpienia błędów.

TypeScript sprawdza program pod kątem błędów przed wykonaniem i robi to w oparciu o rodzaje wartości, jest to statyczny kontroler typu.

- ułatwia kontrolę nad aplikacją
- podpowiedzi w edytorze kodu
- pozwala uniknąć wielu częstych błędów
- ułatwia czytanie kodu
- szybszy i łatwiejszy refactoring kodu
- bezpieczne zmiany





# Wady Typescript

Na początku projektu korzystanie z Typescript może wiązać się z dodatkowym nakładem pracy, ponieważ istnieje potrzeba tworzenia i pilnowania typów.

- pisanie kodu wymaga też pisania typów
- pewne patterny mogą być trudne do otypowania
- błędy które mogą być czasem trudne do rozszyfrowania
- dodatkowa konfiguracja przy starcie projektu
- typowanie nie występuje w "runtime" czyli w trakcie działania aplikacji



# Typescript - typy

## Wbudowane podstawowe typy znane z JS

- string
- number
- boolean
- array
- void
- null
- undefined

## Dodatkowo TypeScript oferuje również typy bardziej zaawansowane:

- any
- unknown
- never
- enum
- tuple



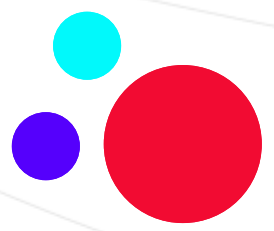
# TypeScript - deklaracja typu

TypeScript zapewnia dwa sposoby tworzenia niestandardowych typów danych – są to aliasy typów i interfejsy.

Aliaszy typów i interfejsy są bardzo podobne i w wielu przypadkach możesz wybierać między nimi swobodnie.

```
type Person = {  
  name: string;  
  surname: string;  
  email: string;  
  age: number;  
  isActive: boolean;  
}
```

```
interface Person {  
  name: string;  
  surname: string;  
  email: string;  
  age: number;  
  isActive: boolean;  
}
```



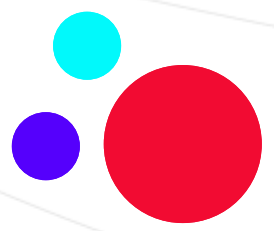
# Typescript - boolean, number, string

```
// boolean
const isAdmin: boolean = true;
const isOpen: boolean = false;

// number
const age: number = 18;
const count: number = 0;
const infinity: number = Infinity;
const binary: number = 0b1010101;

// string
const x:string = 'Hello';
const y:string = "world";
// string Literal Type
type GroupBy = 'second'/'minute'/'hour'/'day';
const groupBy: GroupBy = 'day';
```





# Typescript - array

Podobnie jak w JS, w TS możemy operować na tablicach wartości.

Typ tablicowy możemy zapisać na dwa sposoby:

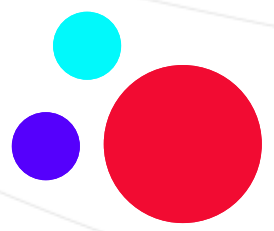
```
// wykorzystujemy ogólny typ tablicy Array<elemType>
const arr1: Array<number> = [1, 2, 3];

// używasz typu elementów 'numbers' po którym następuje oznaczenie tablicy '[]'
const arr2: number[] = [1, 2, 3];

const values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];

// definicja typu tablicy
type Fruits = string[];

// użycie wcześniej zdefiniowanego typu
const fruits: Fruits = ['Apple', 'Orange', 'Banana'];
```



# Typescript - alias funkcji

Możliwe jest również zdefiniowanie typu oznaczającego funkcję.

Jest to bardzo przydatne przy opisywaniu definicji callbacków przekazywanych do funkcji.

```
// parametr s jest stringiem a funkcja reverse zwraca również string
function reverse(s: string): string {
  return s.split("").reverse().join("");
}

// definicja typu funkcji
// parametrem jest obiekt User a funkcja zwraca boolean
type UserCallback = (user: User) => boolean;

// funkcja fetchUser przyjmuje parametr callback typu UserCallback
// funkcja fetchUser jest voidem bo nic nie zwraca
function fetchUser(callback: UserCallback): void {
  callback({ name: 'Dariusz' })
}
```



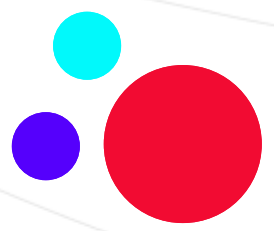
# Typescript - any

Możliwe jest zdefiniowanie typu którego nie będziemy w stanie określić.

Zmienne typu **any** mogą przyjmować dowolne wartości.  
Z reguły będziemy starać się unikać używania tego typu

```
function displayEl(el: any): void {  
  console.log(el)  
}
```

```
displayEl(1);  
displayEl('string');  
displayEl({ name: 'Adam' });  
displayEl(true);  
displayEl(() => false);
```



# Typescript - void

Ten typ oznacza „brak wartości”. Powszechnie używa się go do oznaczania funkcji, które nic nie zwracają.

Deklarowanie zmiennych typu void nie jest przydatne, ponieważ można tylko przypisać null i undefined do nich.

```
function displayEl(el: any): void {  
    console.log(el)  
}  
  
function warnUser(): void {  
    console.log("This is my warning message");  
}  
  
function showAlert(text:string):void {  
    window.alert(text);  
}
```





# Typescript - tuple (krotka)

Tuple to wyrażenie tablicy ze stałą liczbą elementów, których typy są znane, ale nie muszą być takie same.  
Na przykład możesz chcieć przedstawić wartość jako parę string i number.

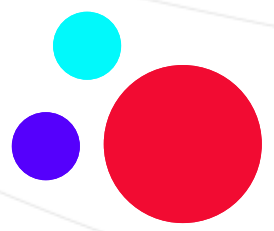
```
const tuple:[number, string] = [1, 's'];  
const x: [string, number] = ["hello", 10];  
  
type state = string;  
type setState = (s: string) => string;  
type React = [state, setState]  
  
const [state, setState]: React = ['', (s) => s]
```



# Typecript - enum

Enumeracja to zbiór nazwanych wartości.  
Znany z wielu innych języków takich jak Java, C#, C++.

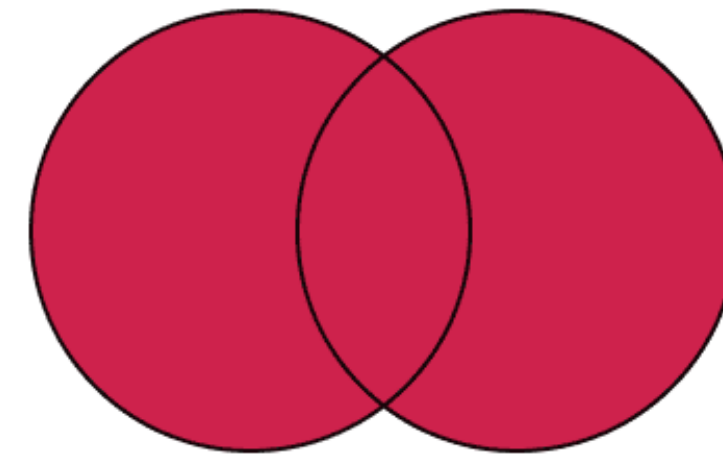
```
enum UserType {  
    SuperAdmin,  
    Admin,  
    Manager,  
    User  
};  
  
const user:UserType = UserType.SuperAdmin; // 0  
  
// Domyślnie elementy enumeracji są numerowane od zera, ale można to zmienić  
  
enum UserType2 {  
    SuperAdmin = 5,  
    Admin = 4,  
    Manager = 3,  
    User = 123  
};  
  
const user2:UserType2 = UserType2.SuperAdmin; // 5
```



# Typescript - Union vs Intersection

## Union

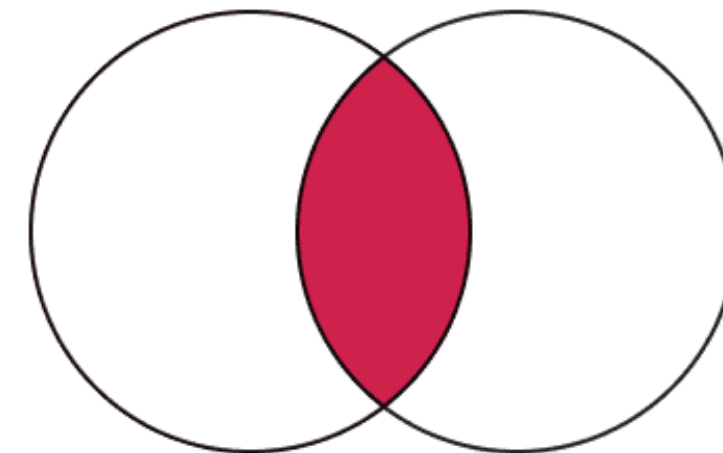
type C = type A | type B



A | B

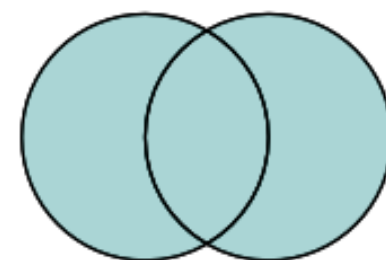
## Intersection

type C = type A & type B

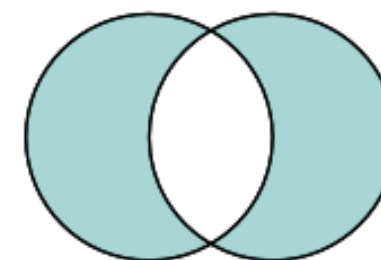


A & B

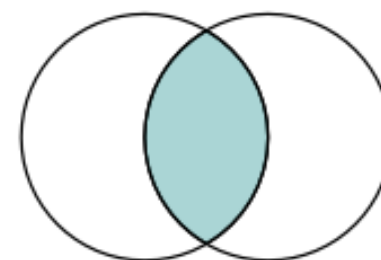
## Union



## Difference



## Intersection



# THANK YOU FOR YOUR ATTENTION

[infoShareAcademy.com](https://infoShareAcademy.com)