TYPESCRIPT

Hello

Kamil Richert

Senior Software Engineer at Atlassian

TYPESCRIPT

TypeScript rozszerza JavaScript o możliwość typowania statycznego. Dzięki czemu jesteśmy w stanie wyłapać więcej błędów zanim oprogramowanie trafi na produkcję.

Co nam daje TypeScript?

1. Zapobiega błędom i ułatwia debugowanie kodu
2. Ułatwia kontrolę nad aplikacją (refaktoryzację i utrzymywanie kodu)
3. Większa czytelność kodu
4. Wymusza mniejsze funkcje
5. Podpowiedzi w edytorze kodu
6. Sprawdza poprawność typów podczas kompilacji
7. Każdy kod JS jest prawidłowym kodem TS
8. TS finalnie jest kompilowany do JS

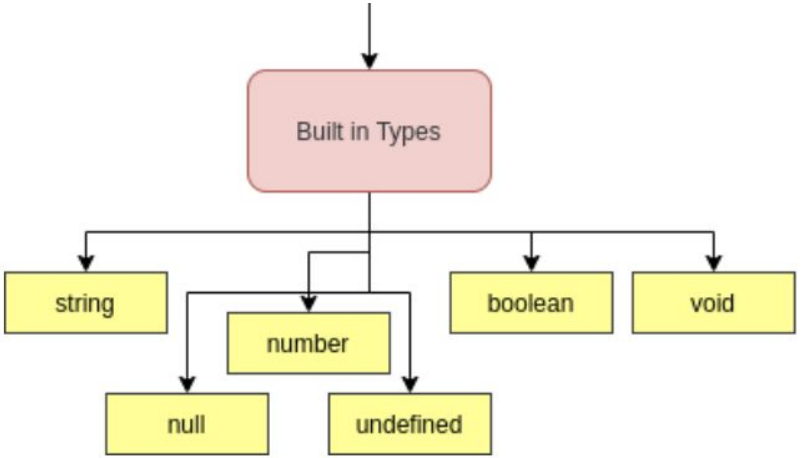
Jakie narzuca utrudnienia

1. Zwalnia czas wydawania oprogramowania
2. Dodatkowa konfiguracja przy starcie projektu

Typy zmiennych

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

- never
- any
- unknown



Deklaracja zmiennych

Typy zmiennych można definiować za pomocą dwukropka po nazwie zmiennej, np.:

```
let liczba: number = 10;
let napis: string = "Hello, world!";
let prawda: boolean = true;
let dowolnyTyp: any = "To może być cokolwiek!";
let brakWartości: void = undefined;
let pustaWartość: null = null;
let niezdefiniowanaWartość: undefined = undefined;
```

Inferencja typów

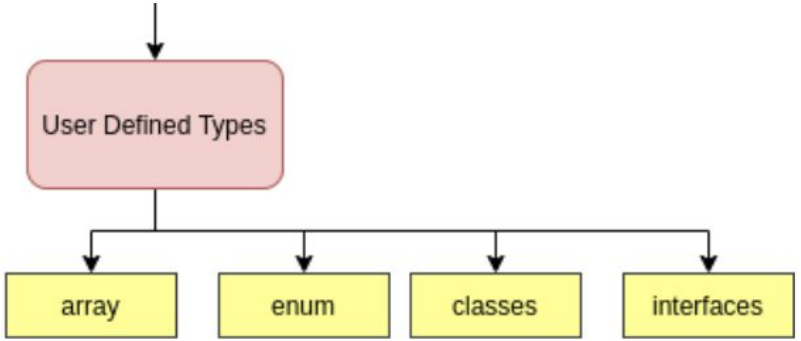
Przy deklaracji wartości TS zapamiętuje typ, także nie musimy za każdy razem typować ręcznie

```
let trainerName = 'Ash'; // type name!
let maxPokemonCount = 6; // type number!
let gonnaCatchThemAll = true // type boolean;
```

```
let trainerName: string
Type 'boolean' is not assignable to type 'string'. (2322)
Peek Problem (Alt+F8)    No quick fixes available
trainerName = false;
maxPokemonCount = '3';
```

Typy złożone

- array
- enum
- classes
- types
- interfaces



Typowanie funkcji

Możemy typować zarówno deklaracje funkcji, jak i funkcje przypisywane do zmiennych. Przy funkcjach strzałkowych obowiązkowe są nawiasy

```
function catchPokemon(pokemonName: string, pokeball: Pokeball): boolean {
    // try to catch

    return true
}


const catchPokemon = (pokemonName: string, pokeball: Pokeball): boolean => {
    // try to catch
    return true
}
```

Jeśli funkcja nic nie zwraca, używamy typu *void*

Słowo kluczowe as

```
interface Pokeball {
    name: string;
    chanceRate: number;
    rarity: PokeballRarity;
}

let ultraball: Pokeball = {};

let greatball: Pokeball = {} as Pokeball;
```

```
let ultraball: Pokeball

Type '{}' is missing the following properties from type
'Pokeball': name, chanceRate, rarity (2739)

Peek Problem (Alt+F8)    No quick fixes available
```

Słowo kluczowe **as** pozwoli nam powiedzieć TypeScriptowi, że dany obiekt na na pewno będize danego typu nawet jeśli teraz tak nie wygląda

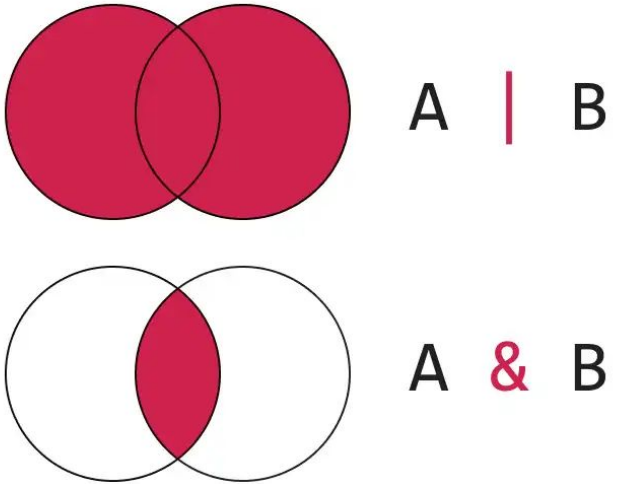
Union & Intersection

Union

type C = type A | type B

Intersection

type C = type A & type B



Union & Intersection

W TypeScript unie i intersekcje są dwoma sposobami łączenia typów, aby utworzyć nowe typy.

Unia to połączenie dwóch lub więcej typów, w wyniku czego otrzymujemy nowy typ, który może być jednym z tych typów. Używa się operatora | do łączenia typów w unię.

```
type Status = 'open' | 'closed';
type Priority = 'low' | 'medium' | 'high';
```

Union & Intersection

Intersekcja to połączenie dwóch lub więcej typów, w wyniku czego otrzymujemy nowy typ, który zawiera wszystkie właściwości i metody z tych typów. Używa się operatora & do łączenia typów w intersekcję.

```
type Person = {
  name: string;
  age: number;
};
type Employee = {
  position: string;
};
type Manager = Person & Employee;
```

Tablice

Aby zadeklarować typ tablicy w TypeScript, możesz użyć nawiasów kwadratowych [] po typie elementów tablicy.

```
let numbers: number[] = [1, 2, 3];
let strings: string[] = ['hello', 'world'];
type Person = {
  name: string;
  age: number;
};

let people: Person[] = [
  { name: 'John', age: 30 },
  { name: 'Jane', age: 25 },
];
```

Enum

Enum w TypeScript to typ danych, który pozwala na definiowanie zestawu stałych wartości oznaczonych nazwami. Enumy są przydatne w przypadkach, gdy chcemy używać stałych wartości, które można łatwo zrozumieć i używać w kodzie.

```
enum Color {
  Red = 1,
  Green = 2,
  Blue = 3,
}

let color: Color = Color.Green;
console.log(color); // 2
```

Interfejsy

Interfejsy to szablony, które definiują strukturę obiektów. Pozwalają one na określenie wymaganych pól i metod, jakie musi posiadać dany obiekt, aby spełniał określone wymagania. Interfejsy można definiować za pomocą słowa kluczowego interface, np.:

```
interface Osoba {
  imie: string;
  nazwisko: string;
  wiek: number;
}
```

Interfejs vs type

Główne różnice między type a interface to:

- type pozwala na tworzenie typów niestandardowych, takich jak np. unie, aliasów, literałów, a interface pozwala na definiowanie tylko obiektów lub klas.
- interface umożliwia dziedziczenie za pomocą słowa kluczowego extends, podczas gdy type nie pozwala na dziedziczenie.
- type można użyć wraz z innymi typami, takimi jak union czy intersection, podczas gdy interface nie można.
- interface po ponownej deklaracji jest scalany z poprzednią deklaracją

Interfejs vs type

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

type UserID = string | number;

type Admin = User & {
  permissions: string[];
};
```

Interfejs vs type

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

interface IAdmin extends IUser {
  permissions: string[];
}
```

Klasy

- public
- private
- protected
- static
- readonly
- getter
- setter

```
class Osoba {
  imie: string;
  nazwisko: string;
  wiek: number;

  constructor(imie: string, nazwisko: string, wiek:
number) {
    this.imie = imie;
    this.nazwisko = nazwisko;
    this.wiek = wiek;
  }

  przedstawSie(): void {
    console.log(`Nazywam się ${this.imie}
${this.nazwisko} i mam ${this.wiek} lat.`);
  }
}

let marek = new Osoba("Marek", "Kowalski", 30);
marek.przedstawSie(); // Nazywam się Marek Kowalski i
mam 30 lat.
```

Public

```
class Pokemon {
    constructor(public name: string,
                public pokeType: PokemonType,
                public attack: PokemonAttack[]) {}
}

const pikachu = new Pokemon('Pikachu', 'lighting', []);


pikachu.
```

```
attack                    (property) Pokemon.at…
name
pokeType
```

Public

```
class Pokemon {
    constructor(public name: string,
                private pokeType: PokemonType,
                private attack: PokemonAttack[]) {}
}

const pikachu = new Pokemon('Pikachu', 'lighting', []);

pikachu.
```

```
name                                (property) Pokemon.na…
```

Protected

```
class Pokemon {
    constructor(public readonly name: string,
                private pokeType: PokemonType) {}
}

class LightingPokemon extends Pokemon {
    constructor(public readonly name: string) {
        super(name, 'lighting');
    }

    sayHello(): void {
        console.log(`I am ${this.pokeType} pokemon`);
    }
}

const pikachu = new LightingPokemon('Pikachu');

pikachu.
```

```
name                    (property) LightingPo…
sayHello
```

```
class Pokemon {
    constructor(public readonly name: string,
                protected pokeType: PokemonType) {}
}

class LightingPokemon extends Pokemon {
    constructor(public readonly name: string) {
        super(name, 'lighting');
    }

    sayHello(): void {
        console.log(`I am ${this.pokeType} pokemon`);
    }
}

const pikachu = new LightingPokemon('Pikachu');

pikachu.
```

```
name                    (property) Lig
sayHello
```

Static

```
class Pokemon {
    static currentWorld = 'Kanto';

    constructor(public name: string,
                private pokeType: PokemonType,
                private attack: PokemonAttack[]) {}
}

const pikachu = new Pokemon('Pikachu', 'lighting', []);

pikachu.currentWorld;

Pokemon.currentWorld;
```

Readonly

```
class Pokemon {
    static currentWorld = 'Kanto';

    constructor(public readonly name: string,
                private pokeType: PokemonType,
                private attack: PokemonAttack[]) {}
}

const pikachu = new Pokemon('Pikachu', 'lighting', []);

pikachu.name = 'Raichu';
```

Getter, setter

```
class Pokemon {
    constructor(private _name: string) {}

    get name(): string {
        return this.name.toUpperCase();
    }

    set name(value: string) {
        // sideeffects
        console.log('changing name');

        this._name = value;
    }
}

const pikachu = new Pokemon('pikachu');

pikachu.name;           // fires getter function
pikachu.name = 'ditto'; // fires setter function
```

Typy generyczne

Jeżeli chcemy stworzyć typ, który ma być reużywalny, ale zmienia się na przykład tylko 1 element to możemy wykorzystać generyczny typ.

```
identity<Type>(arg: Type): Type {
 return arg;
}
```

Narzędzia typowania

Required
Partial
Pick
Omit
…

https://www.typescriptlang.org/docs/handbook/utility-types.html

```
type PartialCars = Partial<Cars>;

type RequiredUser = Required<User>;

type PickCars = Pick<Cars, 'model'>

type OmitCars = Omit<Cars, 'id'>
```

tsconfig

Aby skonfigurować projekt TypeScript, należy utworzyć plik konfiguracyjny. Po prawej strony przykładowy plik.

tsconfig.json to nic innego to ustawienia jak ma działać kompilator tsc, na co ma zwracać uwagę itp.

Reguły TS są bardzo przejrzyście opisane i udokumentowane

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "noFallthroughCasesInSwitch": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx"
  },
  "include": [
    "src"
  ]
}
```

Dzięki

Znajdziecie mnie:
https://www.linkedin.com/in/kamil-richert/
https://github.com/krichert