Deklaracja zmiennych

W TypeScript występuje **statyczny mechanizm typowania.**

Trzy główne typy prymitywne:

1) **string –** wartości tekstowe

   let nazwisko: string = “Kowalski”;

2) **number –** liczby całkowite i wartości zmiennoprzecinkowe

   let wartosc: number = 17.5;

3) **boolean –** wartości prawda lub fałsz

   let czyPrawda: boolean = false;

Można tu jeszcze wspomnieć o **bigint**, która przechowuje liczby całkowite i wartości zmiennoprzecinkowe, ale umożliwia używanie większych liczb ujemnych i dodatnich niż typ number.

Typy specjalne

- **any**, który umożliwia przypisywanie dowolnego typu zmiennym.

  ```
  let cokolwiek: any;
  cokolwiek = "szkoła";
  console.log(typeof cokolwiek + ", " + cokolwiek);        // string, szkoła
  cokolwiek = 64;
  console.log(typeof cokolwiek + ", " + cokolwiek);        // number, 64
  ```

- **unknown** – bezpieczniejsza alternatywa dla any. TypeScript zapobiega używaniu nieznanych typów.
- **null** w TypeScript oznacza celowy brak wartości i jest używany, gdy chcemy wyraźnie wskazać, że wartość jest pusta.
- **undefined** oznacza, że zmienna została zadeklarowana, ale nie przypisano jej jeszcze żadnej wartości.

Do deklaracji zmiennej można zamiast *let* użyć *var*, jednak nie jest to zalecana praktyka.

Przy deklaracji zamiast let możemy użyć **const** (stała). Zmienna zadeklarowana przy użyciu const

nie może być przypisana ponownie. Wartość przypisana podczas inicjalizacji pozostaje stała.

const y: number = 5;
y = 10; // Error

Inferencja (dedukcja) typów

W wielu miejscach kompilator jest w stanie „domyślić się” jaki jest typ zmiennej. Przykładowo w poniższym zapisie określanie typu jest zbędne:

let nazwisko: string = “Kowalski”;

Jeżeli od razu przypisujemy wartość do zmiennej, w większości przypadków typ powinien zostać rozpoznany sam, tak więc ten zapis jest równoważny:

let nazwisko = “Kowalski”;

TypeScript (podobnie jak JavaScript) uwzględnia **wielkość liter** (jest *case-sensitive*). Przykład:

let name = "Alice";
let Name = "Bob";
console.log(name); // "Alice"
console.log(Name); // "Bob"
let Name = "Vinicius"; // Error

Komentarze w języku TypeScript

Jednoliniowy: *// treść komentarza.*

Wieloliniowy: */* treść komentarza */*

Typy złożone

1) **array –** tablica

   const imiona: string[] = ["Jan", "Filip", "Adam"];
   const imiona2: Array<string> = ["Tadeusz", "Gilberta", "Albert"];
   const nazwiska = ["Kowalski", "Nowak", "Tryla"];
   const wiek: number[] = [45, 50, 19];
   console.log(nazwiska[0] + ", " + nazwiska[1] + ", " + nazwiska[2]);
   // Kowalski, Nowak, Tryla

   Wybrane metody tablicy:

   - push(element) – dołącza określony element na końcu tablicy
   - pop() – usuwa i zwraca ostatni element w tablicy
   - shift() – usuwa i zwraca pierwszy element w tablicy
   - unshift() – wstawia nowy element na początku tablicy
   - sort() – zwraca tablicę, której elementy są posortowane w kolejności rosnącej
   - reverse() – zwraca tablicę, której elementy są posortowane w kolejności malejącej

- concat(nazwa_dołączanej_tablicy) – zwraca tablicę, która powstaje w wyniku połączenia tablicy z tą podaną w argumencie
- join(separator) – zwraca ciąg tekstowy wszystkich wartości przechowywanych w tablicy; poszczególne elementy są od siebie oddzielone separatorem
- slice(start, stop) – zwraca fragment tablicy; start to indeks początku, stop to indeks końca

2) **tuple** – krotka. Jest to tablica o stałej długości, której poszczególne elementy mogą być różnych typów.

```
const krotka: [string, number] = ["Jan Kowalski ", 40]
console.log(krotka[0] + ", " + krotka[1]);
```

3) **enum –** wyliczenie. Jest to specjalna "klasa", która reprezentuje grupę stałych. Wyliczenie pozwala nadawać zbiorom wartości bardziej przyjazne nazwy.

```
enum kolory {
    czerwony,
    zielony,
    niebieski
}
let niebieski = kolory.niebieski;
let niebieski2 = kolory[2];
```

4) **object** – zbiór opisujących go zmiennych i funkcji. W TypeScript nie trzeba tworzyć klasy aby utworzyć obiekt.

```
const person = {
        name: "Alice",
        age: 30,
        greet: function() {
            console.log(`Hello, my name is ${this.name}`);
        }
};
person.greet();  // "Hello, my name is Alice"
```

Definiowanie funkcji

Funkcja jest zbiorem instrukcji, które wykonują określone zadanie.

Deklaracja funkcji składa się ze słowa kluczowego *function* oraz:

- **nazwy funkcji**
- **listy parametrów** wraz z określeniem ich typu, umieszczonych w nawiasach okrągłych i oddzielonych przecinkami
- **instrukcji**, które zostaną wykonane po wywołaniu funkcji, umieszczonej w nawiasach klamrowych

Uwaga: o parametrach mówimy, gdy funkcję tworzymy, o argumentach zaś – gdy ją wywołujemy.

Przykłady funkcji:

```
function printHello(): void {
    console.log('Hello!');
}
printHello();   // Hello!

function powitanie(tresc: string): void {
    console.log(`Witaj, ${tresc}`);
}
powitanie('Cezary'); // Witaj, Cezary!
```

TypeScript nie pozwala na zdefiniowanie funkcji o już istniejącej nazwie, a także nie osługuje tzw. przeciążania funkcji.

Funkcja suma:

```
function suma(a: number, b: number): number {
    return a + b;
}
console.log(`Suma liczb 2 i 7 wynosi ${suma(2,7)}`);
```

Funkcja suma z **parametrem opcjonalnym**:

```
function suma(a: number, b: number, c?: number) {
    return a + b + (c || 0);     // jeśli parametr c nie zostanie przekazany, to użyta zostanie wartość 0
}
console.log(suma(2,5))
```

Przy wywołaniu funkcji istotna jest kolejność podawanych argumentów.

TypeScript obsługuje domyślne wartości dla argumentów:

```
function powitaj(imie: string, przywitanie: string = "Cześć"): string {
    return `${przywitanie}, ${imie}`;
}
powitaj("Jan");         // Cześć, Jan
powitaj("Jan", "Witaj");        // Witaj, Jan
```

Funkcja wariadyczna to taka, która może przyjąć dowolną liczbę argumentów.

```
function powitanie(...imiona: string[]): string {
    return `Witajcie ${imiona.join(", ")}`;
}
console.log(powitanie("Cezary", "Tadeusz", "Marcin")); // Witajcie Cezary, Tadeusz, Marcin
```

Parametrem tej funkcji jest **tablica** (określana także jako **parametr resztowy**).

Jeśli funkcja ma kilka parametrów, tablicę umieszczamy na końcu:

```
function powitanie(imie: string, ...dane: string[]): string {
```

```
    return `Witaj ${imie}, pozostałe dane o tobie: ${dane.join(", ")}`;
}
console.log(powitanie("Cezary", "Gorlice", "Real Madryt"));
// Witaj Cezary, pozostałe dane o tobie: Gorlice, Real Madryt
```

Funkcja anonimowa

Funkcja anonimowa to funkcja bez nazwy, która jest często przypisywana do zmiennej lub przekazywana jako argument do innych funkcji.

```
const suma = function (a: number, b: number): number {
    return a + b;
};
console.log(`Suma liczb 2 i 7 wynosi ${suma(2, 7)}`);
```

Funkcja strzałkowa

Mają bardziej zwięzłą składnię. Jeśli funkcja ma tylko jedno wyrażenie, możemy pominąć nawiasy {} i return, a wynik wyrażenia zostanie zwrócony automatycznie.

```
const suma = (a: number, b: number): number => a + b;
console.log(`Suma liczb 2 i 7 wynosi ${suma(2, 7)}`);
```

Zadanie

Napisz w TypeScript funkcję, która obliczy i wyświetli w konsoli różnicę, iloczyn i iloraz dwóch liczb.

```
function obliczOperacje(a: number, b: number): void {
    console.log(`Różnica liczb ${a} i ${b} wynosi: ${a-b}`);
    console.log(`Iloczyn liczb ${a} i ${b} wynosi: ${a*b}`);
    console.log(`Iloraz liczb ${a} i ${b} wynosi: ${a/b}`);
}
obliczOperacje(10, 5);
```

Klasy

Klasa to wzór (szablon), na podstawie którego tworzone są obiekty. Posiada pola i metody używane do budowy obiektu.

Obiekt to instancja klasy.

Przykład klasy w TypeScript:

```
class Klient {
   imie!: string;
   nazwisko!: string;
   wiek!: number;
   kontakt!: string[];
}
```

```
const Nowak = new Klient();
Nowak.imie = "Tadeusz";
Nowak.nazwisko = "Nowak";
Nowak.wiek = 45;
Nowak.kontakt = ["123456789", "tadnow@gmail.com"];

console.log(`Imię: ${Nowak.imie}`);
console.log(`Nazwisko: ${Nowak.nazwisko}`);
console.log(`Wiek: ${Nowak.wiek}`);
console.log(`Kontakt: ${Nowak.kontakt.join(", ")}`);
```

Przykład klasy z konstruktorem:

```
class Klient {
    imie: string;
    nazwisko: string;
    wiek: number;
    kontakt: string[];

    // Konstruktor przyjmujący wartości dla każdej właściwości
    constructor(imie: string, nazwisko: string, wiek: number, kontakt: string[]) {
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.wiek = wiek;
        this.kontakt = kontakt;
    }

    // Metoda wyswietl, która wypisuje informacje o kliencie
    wyswietl() {
        console.log(`Imię: ${this.imie}`);
        console.log(`Nazwisko: ${this.nazwisko}`);
        console.log(`Wiek: ${this.wiek}`);
        console.log(`Kontakt: ${this.kontakt.join(", ")}`);
    }
}

// Tworzenie nowej instancji klasy Klient za pomocą konstruktora
const Nowak = new Klient("Tadeusz", "Nowak", 45, ["123456789", "tadnow@gmail.com"]);

// Wywołanie metody wyswietl, aby wypisać informacje o kliencie
Nowak.wyswietl();
```

Hermetyzacja

TypeScript wprowadza modyfikatory dostępu, które pozwalają kontrolować widoczność właściwości i metod:

- public – pola/metody dostępne wszędzie (domyślny modyfikator).

- private – dostępne tylko wewnątrz klasy.
- protected – dostępne w klasie i jej klasach dziedziczących.

Przykład:

```
class Car {
   public brand: string; // publiczne
   private engineNumber: string; // prywatne
   protected year: number; // chronione

   constructor(brand: string, engineNumber: string, year: number) {
      this.brand = brand;
      this.engineNumber = engineNumber;
      this.year = year;
   }

   public getCarInfo(): string {
      return `Brand: ${this.brand}, Year: ${this.year}`;
   }
}
```

Skrócona inicjalizacja właściwości:

```
class Car {
   constructor(
      public brand: string, // publiczna właściwość
      private engineNumber: string, // prywatna właściwość
      protected year: number // chroniona właściwość
   ) {}

   public getCarInfo(): string {
      return `Brand: ${this.brand}, Year: ${this.year}`;
   }
}
```

Gettery i settery:

```
class Car {
   public brand: string; // publiczne
   private engineNumber: string; // prywatne
   protected year: number; // chronione

   constructor(brand: string, engineNumber: string, year: number) {
      this.brand = brand;
      this.engineNumber = engineNumber;
      this.year = year;
   }

   // Getter do pola engineNumber
   public get engineNumber(): string {
```

```
        return this.engineNumber;
    }

    // Setter do pola engineNumber
    public set engineNumber(newEngineNumber: string) {
        if (newEngineNumber.length > 0) {
            this.engineNumber = newEngineNumber;
        } else {
            console.log("Engine number cannot be empty.");
        }
    }

    // Publiczna metoda do pobrania informacji o aucie
    public getCarInfo(): string {
        return `Brand: ${this.brand}, Year: ${this.year}`;
    }
}

// Tworzenie obiektu
const myCar = new Car("Toyota", "ENG12345", 2020);

// Wywołanie metody getCarInfo
console.log(myCar.getCarInfo()); // Brand: Toyota, Year: 2020

// Użycie getterów i setterów
console.log(myCar.engineNumber); // ENG12345

myCar.engine = "NEWENGINE6789"; // Zmiana numeru silnika
console.log(myCar.engine); // Engine Number: NEWENGINE6789

myCar.engine = ""; // Próba przypisania pustego numeru
// Engine number cannot be empty.
```

Elementy oznaczone jako **static** należą do klasy, a nie do jej instancji. Można je wywołać bez tworzenia obiektu. Przykład:

```
class MathUtils {
    static PI = 3.14;

    static calculateCircleArea(radius: number): number {
        return this.PI * radius * radius;
    }
}
console.log(MathUtils.calculateCircleArea(5)); // 78.5
```

Dziedziczenie

Dziedziczenie to jeden z kluczowych mechanizmów programowania obiektowego (OOP), który pozwala tworzyć nowe klasy na podstawie istniejących, dziedzicząc ich właściwości i metody. Dzięki temu możemy pisać bardziej zwięzły i mniej powtarzalny kod.

Podstawy dziedziczenia:

- **Klasa bazowa** (ang. *base class* lub *superclass*) to klasa, która przekazuje swoje składowe klasie pochodnej.
- **Klasa pochodna** (ang. *derived class* lub *subclass*) to klasa, która dziedziczy składowe z klasy bazowej.

W TypeScript dziedziczenie realizuje się za pomocą słowa kluczowego **extends**, który łączy klasę pochodną z klasą bazową.

Przykład:

```
// Klasa bazowa
class Shape {
    public name: string;

    constructor(name: string) {
        this.name = name; // Przypisanie wartości do pola
    }

    // Metoda wspólna dla wszystkich figur
    public describe(): string {
        return `This is a ${this.name}.`;
    }

    // Metoda do obliczania pola (powinna być nadpisana w klasach dziedziczących)
    public getArea(): number {
        return 0;
    }
}

// Klasa Prostokąt (Rectangle) dziedzicząca po klasie Shape
class Rectangle extends Shape {
    public width: number; // Pole specyficzne dla prostokąta
    public height: number; // Pole specyficzne dla prostokąta

    constructor(name: string, width: number, height: number) {
        super(name); // Wywołanie konstruktora klasy bazowej
```

```
        this.width = width; // Przypisanie wartości do pola
        this.height = height; // Przypisanie wartości do pola
    }

    // Implementacja metody getArea
    public getArea(): number {
        return this.width * this.height;
    }
}

// Klasa Koło (Circle) dziedzicząca po klasie Shape
class Circle extends Shape {
    public radius: number; // Pole specyficzne dla koła

    constructor(name: string, radius: number) {
        super(name); // Wywołanie konstruktora klasy bazowej
        this.radius = radius; // Przypisanie wartości do pola
    }

    // Implementacja metody getArea
    public getArea(): number {
        return Math.PI * this.radius * this.radius;
    }
}

const shape = new Shape("neat shape");
console.log(shape.getArea());

const rectangle = new Rectangle("neat rectangle", 10, 5);
console.log(rectangle.describe()); // This is a blue shape.
console.log(`Rectangle area: ${rectangle.getArea()}`); // Rectangle area: 50

const circle = new Circle("neat circle", 7);
console.log(circle.describe()); // This is a red shape.
console.log(`Circle area: ${circle.getArea().toFixed(2)}`); // Circle area: 153.94
```

Klasy abstrakcyjne

Klasa abstrakcyjna działa jako szablon dla innych klas. Nie można utworzyć instancji klasy abstrakcyjnej.

```
abstract class Shape {
    abstract getArea(): number; // Metoda abstrakcyjna (bez implementacji)

    displayArea() {
        console.log(`Area: ${this.getArea()}`);
    }
}
```

```
class Rectangle extends Shape {
    constructor(public width: number, public height: number) {
        super();
    }

    getArea(): number {
        return this.width * this.height;
    }
}

const rect = new Rectangle(10, 20);
rect.displayArea(); // Area: 200
```