**TYPESCRIPT OGOLNE INFORMACJE**

Dobrą praktyką jest oznaczanie jako **private** wszystkiego co tylko się da, tak aby z zewnątrz był dostęp wyłącznie do tych pól i metod, które są potrzebne. JEst to **enkapsulacją** /hermetyzacją.

**SILNE TYPOWANIE-** oznacza, że zmienna o ustalonym typie nie może być użyta tam, gdzie oczekiwany jest inny typ.

**array - Deklarowanie typow tablic. Dwa sposoby, równoznaczne**

**let** list1: number[] = [1, 2, 3];

**let** list2: Array<number> = [1, 2, 3];

**let** person1: [string, number] = ['Jan', 22];

**const** arr1 = [1, 2, 3];//uznal ze to typ number[] wiec arr1.push('lola') error

**const** mixArr = [1, 'dupa', 3]//uzna ze to arr

mixArr.push(true)//blad,bo może być tylko typ z pierwotnej arr,bezznaczenia indx i gdzie

// !!!!! nie da sie wstawiac wartosci do arr zadeklarowanej, ale nie zainicjowanej!!!

**let** arr2: string[]; arr2.push('bro')// Uncaught TypeError: arr2 is undefined

(array z elementami/pochodnymi typu konkretnej klasy bazowej omowione w dziale class)

**Enumy i reverse mapping (z tutorialsteacher.com tez z excela)**

**enum** Color { Red = 5, Blue, Green } **let** c: Color = Color.Blue;

**enum** PrintMedia {

Newspaper = 1,

Newsletter,

Magazine,

Book

}

console.**log**(PrintMedia) // drukuje najpierw cala liste w postaci '1':'Newspaper', '2':"News...,a potem odwrotnie po nazwie i index:)

console.**log**(PrintMedia[2])//Newsletter console.**log**(PrintMedia.Newsletter)//2

**any i unknown**

**let** dupa: unknown = 10; //any nie zglosi bledu mimo,ze wywolam ją jak funk. Lub nieist-niejaca wlasciw itd.(czego nie ma nie odpali ale bez wolania blad) Ale unknown juz tak.

**union type**

**let** kasza: **boolean** | number; kasza = 30; kasza = false;//kasza = 'tak'; // blad

**tuples (krotki) –** na danym indeksie musi być konkretny typ. Sami decydujemy o ich typie i ilości.

let zmienna: [string,number,boolean] = [ ‘tomek’, 55, false]  
//array z tuplami  
let zmienna 2: [string, number][]= [[‘jan’,13],[‘ewa’,17],[‘mirek’,22]]

Możemy uzyskać dostęp do elementów krotki za pomocą indeksu, tak samo jak do tablicy. Indeks zaczyna się od 0. Patrz nizej

let employee: [number, string] = [1, "Steve"];

employee[0]; // returns 1

employee[1]; // returns "Steve"

Możesz dodawać nowe elementy do krotki za pomocą metody push().

employee.push(2, "Bill");   
console.log(employee); //Output: [1, 'Steve', 2, 'Bill'], ale… (zgodnie z krotką [nr,str])  
employee.push(true)//test.ts(4,15): error TS2345: Argument of type 'true' is not assignable to parameter of type 'number | string'.

Krotka jest jak tablica. Możemy więc używać metod dla arr (pop(), concat() itp.).

employee[1] = employee[1].concat(" Jobs");

console.log(employee); //Output: [1, 'Steve Jobs']

**user defined type guard**

**let** mojaZmienna: unknown = 10;

//maImie(param to obj typu any) i zwraca obj, ktory zawiera wlasciwosc name typu string

**function** maImie(obj: any): obj is { name: string } {

**return** !!obj && //zwroci obj i ..

**typeof** obj === 'object' && // ten obiekt bedzie typu object i...

"name" **in** obj // w obj istnieje parametr name

}

**if** (maImie(mojaZmienna)) {

console.**log**(mojaZmienna.name)

} //(mojaZmienna as string).toUpperCase();

**funkcje**

**function** dodaj(num1: number, num2?: number) {

**if** (num2) **return** num1 + num2;//jesli istnieje

**else** **return** num1;

}// ? = opcj. Param., da undefined,dlatego musimy zrobic instr. warunkowa wtedy

**let** test0 = dodaj(5, 10);//15 console.**log**("test0:", test0)

let test = dodaj(5, "ddd")//podkresla z bledem

**let** test2 = dodaj(5); console.**log**("test2:", test2)//5

//ta sama funkcja tylko zamiast (?)wstawiam defaultowa wartosc w przypadku nie wpisania

**function** dodajX(num1: number, num2: number = 99) {

**if** (num2) **return** num1 + num2;//jesli istnieje

**else** **return** num1;

}

**let** test3 = dodajX(5) console.**log**("test3:", test3) //104

**const** add = (a: number, b: number, c: number | string = 10) => {

console.**log**(a + b)

console.**log**(c)

}

add(5, 10, "17") //gdy c jest opcjo to=c?, a gdy ma default value(10) to pisze ją

**const** minus = (a: number, b: number) => {

**return** a + b;

}

**let** wynik = minus(10, 20);//wynik od razu tez ma typowana wartosc, przypisana do typu jaka zwracana jest w minusie.

wynik = 5;// ok, ale.. wynik = "100"// juz nie gra,blad bo string:)

**let** greet: (a: string, b: string) => **void**;//a,b=nazwy param.,a w f.nazywam jak chce

**let** calc: (a: number, b: number, c: string) => number;

calc = (numOne: number, numTwo: number, action: string) => {

**if** (action === 'dodaj') {

**return** numOne + numTwo;

} **else** {

**return** numOne - numTwo;

}

}

console.**log**(calc(4, 3, 'odejmij'))//1 console.**log**(calc(4, 3, 'dodaj'))//4

**//gdy parametrem funkcji jest obiekt**

// Należy pamiętać, że nazwa parametru jest wymagana. Przy przykladowej funkcji gdy parametr okreslimy jedynie jako:(string) => ...;oznacza „funkcję z parametrem o nazwie string typu any”!

**let** logDetails: (obj: { name: string, age: number }) => **void**;

//obj to nazwa param. W js ten zapis by wygladal nastepujaco: logDetails(obj)=> {...

logDetails = (ninja: { name: string, age: number }) => {

console.**log**(`${ninja.name} is ${ninja.age} years old

}`); //ninja nie jest potrzebny,ale mozemy sie odwolywac latwiej do param.

//ale to samo mozna zrobic tez z aliasem np.

**let** logDetails2: (obj: { name: string, age: number }) => **void**;

type person = { name: string, age: number };

logDetails2 = (ninja: person) => {

console.**log**(`Sensej ${ninja.name}-san ma ${ninja.age} lat`)

}

console.**log**(logDetails2({ name: 'Marcin', age: 33 }))

**obiekty**

**function** fullName(person: { firstName: string, lastName: string }) {

console.**log**(`${person.firstName} ${person.lastName}`)

}

**const** p = {

firstName: 'Bruce',

lastName: 'Wayne'

} console.**log**(fullName(p))

**Class(dziedziczenie) i access modifiers(public,priv,prot)**

**class** Employee {

Dziedziczenie WIELOPOZIOMOWE opisane po generykach

**protected** employeeName: string;

constructor(name: string) {

**this**.employeeName = name;

}

greet() {

console.**log**(`Good Morning ${**this**.employeeName}!`);

}

}

**const** employee1 = **new** Employee('Janusz'); employee1.greet()

//console.log(employee1.employeeName)//blad przy private i protected bo wywoluje z zew.

**class** Manager **extends** Employee {//dzidzicze z Employee

constructor(managerName: string) {

**super**(managerName);//super(delegowane wlasc.)

}

delegateWork() {//tworze swoja metode

console.**log**('Manager delegating tasks')

console.**log**(**this**.employeeName)//zadziala z protected, a z private juz nie

}

}

**const** manager1 = **new** Manager('Ryszard');

manager1.greet();//metoda dziedziczona manager1.delegateWork()// metoda wlasna

console.log(employee1.employeeName)//dziedzicz. właśc. da blad bo employeeName to privat

**Array z elementami konkretnego typu classy/lub jej pochodnymi**

(np.Zakładając ze mam klase bazowa Animals, a jej pochodnymi sa wszystkie klasy gatunkow)

**const** arrZeZwierzetami: Array<Pies | Koza | Malpa> = [];//Moge użyć tzw.union type:

Jednak to tutaj niepotrzebne, trudne w utrzymaniu, gdyż przy tworzeniu nowej klasy pochodnej musielibyśmy ją dopisać. Możemy zadeklarować taką tablicę, jako przechowującą obiekty typu Animal(classy bazowej), gdyż wszystkie zwierzęta po tej klasie dziedzi:

**const** arr: Array<Animal> = [];

**const** pies = **new** Pies('Bona', 'Black')

**const** koza = **new** Koza("burek")

**const** mapla = **new** Malpa('łoś')

**const** mojaListaZwierzakow = [pies,koza,malpa]

mojaListaZwierzakow.**forEach**(el => el.giveVoice())

/\*Musimy być jednak świadomi, że w ten sposób ograniczamy sobie niejako możliwości wywoływania metod na obiektach z tej tablicy wyłącznie do metod zadeklarowanych w klasie Animal. Przykładowo nie mamy dostępu do metody dodanej dodatkowo w klasie Dog:\*/

**Intersection type**

/\*Intersection type jest blisko związany z union type, ale pozwala na opisanie typu, który ma cechy kilku typów na raz. Najczęściej wykorzystywany jest z interfejsami. Wyobraźmy sobie, że chcemy stworzyć funkcję, która oczekuje objektu będącego na raz typu interface Serializable i Drawable:\*/

**function** mojaFunkcja(obiekt:Serializable & Drawable) {

// obiekt na pewno ma metody toJSON i draw ktore sa z typowanych interfacow!

}

**Interface**

/\*Poza metodami i polami, w interfejsach możliwe jest również zadeklarowanie konstruktorów, index signatures i wywołań(callable interface).\*/

**interface** Person {

firstName: string,

lastName?: string, //opcjonalnie, daje undefined ofc

}

**function** fullName2(person: Person) {//to samo co powyzej ale z interfacem

console.**log**(`Interface: ${person.firstName} ${person.lastName}`)

}

**const** yy = {

firstName: "Jan", // lastName: "matejko" //opcjonalnie

}

fullName2(yy)

**interface** User { //Możesz jawnie opisać kształt tego obiektu za pomocą interface deklaracji:

name: string,

id: number,

}

**const** user: User = {// tworze usera,ktory jest typu Usera, tzn jak interface User

name: 'Michał',//gdy podam inny typ value,lub inna nazwe keysa to tez blad!

id: 0, }

Możesz używać interfejsów do opisywania parametrów i zwracania wartości do funkcji:

function deleteUser(user: User) { //typowanie parametrow

// ...

}

function getAdminUser(): User { //typowanie zwracanej wartosci

//...

}

**interface** Serializable { //typ, wg ktorego obiekt/classa zawiera we wlasciwosciach funkc. toJson():stirng;

toJson(): string;

}

**function** naszaFunkcja(obiekt: Serializable) {

obiekt.toJson();

}

console.**log**(naszaFunkcja(user1))

**interface - opcjonalne wlasciiwosci**

**interface** Osoba {

name: string;

age: number;

[key: string]: any;//moze byc opcj.kolejny [keys typu string] z value typu string

}

**const** pers1: Osoba = {//jest ok mimo ze nie ma 3 parametru,bo ten jest opcjonalny

name: "Heniek",

age: 47,

}

**const** pers2: Osoba = {//jest ok, mimo ze ma 3 parametry,bo ten jest opcjonalny

name: "Bogdan",

age: 33,

job: 'doctor',

}

**interface zad NetNinja**

**interface** isPerson { //szablon wymusza ksztalt obiektu. Nie tworzy go jak klasy!

name: string,

age: number,

speak(a: string): **void**;//parametry w interface moga byc nazywane jakkolwiek

spend(a: number): number,

}

**const** ja: isPerson = { //i tworze obiekt, o typie PERSON

name: 'Arek',

age: 36,

speak(text: string): **void** {

console.**log**(text)

},

spend(amount: number): number {

console.**log**('I spent ', amount);

**return** amount;

},

}

**const** pozdrowOsobe = (osoba: isPerson) => {

console.**log**('hej ', osoba.name)

}

pozdrowOsobe(ja) //wpisujac inny obiet wyskoczy blad ze nie pasuje do definiowanego typu

**classy z interfacem**

**class** PlayerAccount { //sama klasa.oczywiscie napisana w stylu TS

name: string;

id: number;

constructor(name: string, id: number) {

**this**.name = name;

**this**.id = id;

}

}

**interface** Player {

name: string;

id: number;

}

**const** gracz: Player = **new** PlayerAccount('Marian', 8)

**interface** Serializable { //typ, wg ktorego obiekt/classa zawiera we wlasciwosciach funkc. toJson():stirng;

toJson(): string;

}

**class** User **implements** Serializable {

toJson(): string {

**return** '{5}'

}

}

**const** user1 = **new** User();

**// Implementacja wielu interfejsów jest możliwa -**Kolejne implementowane interfejsy wystarczy napisać po przecinku:\*/

**class** User9 **implements** Serializable, Drawable { //nazwy Interfacow Po przecinku

toJSON() {

**return** '{}';

}

draw(color: string) {

console.**log**(color);

}

}

**Aliasy typów niestandardowych (2 przyklady)**

Let sampleUser= { username: string, points: number }; //przykladowa zmienna typowana. Tak uzyta tylko raz

**type** UserData = { username: string, points: number }; // gdzie **type** to slowo kluczowe tworzące **alias**, a **userData**

nazwa aliasu, a po prawej stronie nasz niestandardowy typ. Nie potrzebujemy wtedy w ogole typowania z 1 linijki. A możemy korzystac w wielu przypadkach.

let sampleUser: UserData; // deklaruję zmienną typu niestandardowego aliasu UserData.  
sampeUser= { username: „Janek”, points: 10}; // przypisuje wartość do zmiennej

Natomiast w poniższym przykładzie tworzymy alias dla niestandardowego typu funkcji.

type mojAliasFunkcji = (x: number, y: number) => number; //uzywam arrow funt. Do zdefiniowania typu funkcji

let dodajNumery: mojAliasFunkcji; //………., która ma przyjąć 2 parametry typu number i zwroci wartość typu number

dodajNumery = function (a: number, b: number): number { // przypisuję wartość do zmiennej

return a+b;

};

**Interface vs Alias –**

|  |  |
| --- | --- |
| **Interface** opisuje jak ma wyglądać kształt **obiektu** | **Alias** może uwzgledniac typy, ale tez funkcje jako typy. |
| Opisuje jak coś zaimplementować i użyć | Do zapisu typu, które chcemy uzyc (tez wielokrotnie) |

type CatSpecies = 'lion' | 'tabby'; // **alias** typu unii

**interface** CatInterface { //**interface** używa powyższego aliasu do typowania gatunku kota

species: CatSpecies;

speak(): string;

}

**class** Cat **implements** CatInterface { // **classa** korzystajaca z interfaca

constructor(**public** species: CatSpecies) { }

speak() {

**return** **this**.species === 'lion' ? 'ROAR' : 'meeeooow';

}

}

**const** lion = **new** Cat("lion"); console.**log**(lion.speak()); // ROAR

**Aliasy FUNKCJI**

**interface** Userr {

name: string

}

type UserCallback = (user:Userr) => **boolean**;//alias funkcji,w arg podajemy usera typu Userr(tzn z name w stringu)

**function** fetchUser(callback:UserCallback) { } //deklaruje funkcje(dlaczego :UserCallback typuje w parametrze)

// kod użytkownika

**function** fetchUserCallback(user:Userr) {

**if** (user.name === 'Michal') {

**return** true;

}

**return** false;

}

**const** result= fetchUserCallback({ name: 'Michal' })

**//String Literal Type-**

funkcja jako argument może przyjąć jako typ konkretne wartości.

type GroupBy = 'second' | 'minute' | 'hour' | 'day';

function groupRecords(groupBy: GroupBy) {

…

}

Dzięki temu kompilator sam sprawdzi (w miarę możliwości!), czy podana wartość jest prawidłowa. String literal type świetnie sprawdzi się też jako flaga wspomniana w poprzednim akapicie.

**//Inferencja typów-** Wnioskowanie, jaki typ ma dany element;

**function** fn(b: **boolean**) {

**if** (b === true) {

**return** 1;

} **else** {

**return** 2;

}

}

//Ta funkcja zwraca liczbę i jest to ewidentne. TypeScript również jest tego pewien i dlatego nie musimy tutaj podawać zwracanego typu. TypeScript inferuje, że to number:

**const** liczba: number = fn(true);//w takim wyp,mogę ale nie musze deklarowac **liczba:number**

const liczba = fn(true); //Znow TS inferuje, że zmienna liczba jest typu number.

// W sytuacjach dwuznacznych TS wyświetli błąd i zmusi do zadeklarowania odpow. typu:

**function** fn2(b: **boolean**): string | number {

**if** (b === true) {

**return** 1;

} **else** {

**return** 'lol';

}

}

//Bez deklaracji string | number otrzymalibyśmy błąd: No best common type exists …

// Inferencja tez zadziala automatycznie bez podania typu

**const** tab3 = [**new** Pies('leszek'), **new** Cow('rafal')]; // Array<Pies|Cow>

//oba powyzsze elementy(pies cow) sa instancjami Animal, ale z inferencji typ tej array wyjdzie Array<Pies|Cow>, a nie Array<Animal>, bo każdy ma dodane swoje wlasne wlasciwoasci. By tak bylo musiałbym sam otypowac zmienna const tab3: Animal= [new..]

**GENERICS**

do tworzenia bloku kodu wielorotnego uzytku, ktorych mozna uzywac z roznymi typami.

1. **Generics Z FUNKCJAMI**

**const** dopiszUID = (obj: object) => {

**let** UID = Math.**floor**(Math.**random**() \* 100);

**return** { ...obj, UID }

}

**const** doc1 = { name: "Marek", age: 33 }

**const** doc2 = dopiszUID(doc1)

console.**log**(doc2)//{ name: "Marek", age: 33, UID: ..random X.. }

console.**log**(doc1.name)//"Marek"

console.**log**(doc2.name)// Blad"Property name doesn't exist on this type.."

// Nie da sie wyciagac wlasciwosci ze zwracanych obiektow z funkcji w ten sposob, i wtedy trzeba uzyc generykow..

**const** powiekszOTelefon = <T **extends** {name:string}>(obj: T) => { //T=dowolny,a z extends="dowolny,ktory ma taka wlasc".

**let** tel = Math.**floor**(Math.**random**() \* 10000000);

**return** { ...obj, tel }

}

**const** doc3 = powiekszOTelefon({name: 'borys', age:14})//{name..,age..,tel..}

console.**log**(doc3.name)// borys. Gdybym dal samo <T> to bledu by nie bylo,ale zwroci undf

**T – to nazwa umowna,** ale preferowana. A mozna dowolnie nazwac**.** Ale zgodnie z konwencją używa się dużej litery „T” („T” jak „Typ”) i przesuwa się w dół alfabetu w razie potrzeby.

//Polecenie: zrob funkcje ktora ma miec dwa argumenty typu string lub number. Na koniec ma zwracac tuple, czyli arr z konkretnymi typami na danych pozycjach i tez typu string/nr. Info: Ponieważ Kompilator nie wie jaki będzie dokładny typ to nie zezwala na niektore operacje typowe dla strng/nmbr. Żeby zezwalal trzeba uzyc generykow.

**function** dajTupl<**T**>(a: **T**, b: **T**): [**T**, **T**] {

**return** [a, b]

}

**let** arrStringow = dajTupl<**string**>("Ola", "Ala");

**let** arrNumberow = dajTupl<**number**>(1.53, 3.14)

**let** arStrUpperCase = arrStringow222.**map**(el => el.**toUpperCase**())//ok bez bledu

**let** arNumFloor = arrNumberow222.**map**(el => Math.**floor**(el));//ok bez bledu

// Uwaga! Gdy tworze zmienna korzystajac z tej funkcji typowanej generycznie , ale nie okresle w < > jakiego typu ma byc, to automatycznie wpisujac pierwsz argument A kompilator uznaje, ze takie typy maja byc wszystkich arg. Bo w "szablonie" wszedzie jest **T**, a **T** musi = **T**.

let mixedArr = dajTupl222(5,"ziom") //blad

let mixedArr2 = dajTupl222(true, true); //ok

**Mozna uzyc wiecej niz jednego parametru typowania w funkcji jako argument i zwrot. Np.**

**let** dajTupl = (a, b) => { //Zwykla postac funkcji w formie wyrazenia funkcyjnego

**return** [a, b];

}

//typowanie **<T,U>(a:T,b:U)=>[T,U]** tzn, param.typu **T** i **U**,gdzie arg.a jest **T**,b jest **U**, i zwraca te arg. w arrayu w ustal.kolejnosci(tuplu) **[T,U]** i to typowanie wstawiam do powyzszej funcji, i wychodzi...

**let** dajTupl: <T, U>(a: T, b: U) => [T, U] = (a, b) => {

**return** [a, b]; }

}

**const** mixArr5 = dajTupl("hello", 4)// dzieki temu nie ma bledow i mogą być dowolne

type mojTypeF = <T, U>(a: T, b: U) => [T, U]; // mozna tez uzyc **ALIAS**

**let** dajTupl: mojTypeF = (a, b) => {

**return** [a, b]

} // lub poniżej to samo ale z **INTERFACEM**

**interface** tuplowaFunkcja {

<T, U>(a: T, b: U): [T, U];

}

**let** zwrocTuple: tuplowaFunkcja = (a, b) => {

**return** [a, b]

}

//chce robic interface objektu ktory ma właściwosci a ,b i getTuple() typami generycznmi

**interface** objTuple<T, U> {

a: T;

b: U;

getTuple(): [T, U];

}

**let** tupleObj1: objTuple<number, number> = {

a: 3,

b: 7,

getTuple: **function** () {

**return** [**this**.a, **this**.b]

}

}

**const** tuple1 = tupleObj1.getTuple();//a co, gdy powyzszy obiekt, w funkcji getTuple musi miec podany otypowany argument? Dodaje kolejny param generyczny do interfacu

**interface** objTuple2<T, U, V> {

a: T;

b: U;

getTuple(c: V): [T, U, V];

}

**let** tupleObj3: objTuple2<number, number, string> = {

a: 3,

b: 4,

getTuple(c) {

**return** [**this**.a, **this**.b, c]

},

}

**let** tuple3 = tupleObj3.getTuple("5")

//Wszystko jest ok, pod warunkiem, ze chcemy i wiemy jaki jest typ tego parametru **V**. Jesli nie to musimy ustawic dowolny, ale nie ustawiajac any(omijaj!), tylko nie definiuj go w <generyki> w interfacie, a podajac go przy funkcji przeznaczenia w **<V>** i **zwrocie**,np

**interface** objTuple3<T, U> {

a: T;

b: U;

getTuple<V>(c: V): [T, U, V];

}

A tu kilka generycznych typow, ale w wartości zwracanej

function randomValue<T, U, V>(one: T, two: U, three: V): T | U | V {// This is a tuple

const options: [T, U, V] = [one,two,three];

const rndNum = getRndNumInInclusiveRange(0, 2);//fikcyjna funkcja losowania

return options[rndNum];

}

Const test= randomValue(„dziabong”, 55, true //zwroci arr z losowym argumentem

1. **Generics Z INTERFACEM**

// 1. Tworze interface, ale nie wiem jakiego typu bedzie wartosc wlasciwosci data, dlatego ustawiam typ DOWOLNY, KTORY SPRECYZUJE PODCZAS TWORZENIA OBIEKTU, NA PODSTAWIE TEGO INTERFACE'U

**interface** Resource<T> {

uid: number;

resourceName: string;

data: T;

}

// 2. Tworze OBIEKT, na podstawie interfacu, i tu precyzuje jakiego typu ma byc data;

**const** obiekt1: Resource<object> = {

uid: 22,

resourceName: 'osoba',

data: {name:"Brajan", age:35},

}

**const** obiekt2: Resource<string[]> = {

uid: 22,

resourceName: 'lista zakupów',

data: ["chleb","mleko"],

}

console.**log**(obiekt1, obiekt2)

1. **Generics w classach i interface**

**interface** Secret<T> {

secret: T;

}

**class** Student **implements** Secret<number> {

**public** secret: number;

//public secret: string; blad

constructor(value: number) {

**this**.secret = value;

} }

**D) Dziedziczenie miedzy klasami a typy generyczne**

**class** Dawca<U>{

items: U[];

constructor(...values: U[]) {

**this**.items = values;

}

}

**class** Biorca<T> **extends** Dawca<T>{

dajPierwszyElem(): T {

**return** **this**.items[0];

}

}

**let** litery = **new** Biorca<string>("tak", "jan", "po")

**let** pierwszyEl = litery.dajPierwszyElem()

console.**log**(pierwszyEl.**toUpperCase**())

1. **Dziedziczenie WIELOPOZIOMOWE**

//classa BAZOWA

**class** Animal {

dziedziczyć można tylko po jednej klasie na raz!

constructor(**protected** name: string) { }

giveVoice() {

console.**log**(`Nazywam sie ${**this**.name}`)

}

}

//classa POCHODNA, która dziedziczy z Bazowej

**class** Pies **extends** Animal {

constructor(**protected** name: string) {

**super**(name)

}

giveVoice() {

console.**log**('Szczek szczek!')

**super**.giveVoice();

}

eat() {

console.**log**('mniam mniam')

}

}

console.**log**(**new** Pies("burek").giveVoice())

//classa pochodna z pochodnej ☺

**class** Sznaucer **extends** Pies {

constructor(**protected** name: string, **private** \_color: string) {

**super**(name);

**this**.\_color = \_color;

}

superMoc() {

console.**log**('leze w kuchni')

}

get Kolorek() {

**return** **this**.\_color;

}

}

1. **Klasy ABSTRAKCYJNE - dziedziczenie Cd.**

/\*to klasy bazowe, po których można dziedziczyć, ale nie można ich instancjonować.

Czym się więc różnią od interfejsów? Przede wszystkim klasy abstrakcyjne nie tylko zawierają deklaracje pól i metod, ale mogą też zawierać ich implementacje. Mogą, ale nie muszą. Idealnym kandydatem na klasę abstrakcyjną jest w takim razie wspomniana wcześniej klasa Animal, gdyż jej raczej nie chcemy instancjonować bezpośrednio. Dodajmy też do niej deklarację metody eat, ale bez implementacji (tzw. metodę abstrakcyjną). W ten sposób każda klasa pochodna będzie musiała zaimplementować eat:\*/

**abstract** **class** Animalll {

constructor(**protected** name:string) {}

giveVoice() {

console.**log**(`Nazywam się ${**this**.name}!`);

}

**abstract** eat():**void**;

}

**const** animal = **new** Animalll(''); // blad! Nie mozna tworzyc instancji abstrakcyjnej